VAX LISP/VMS Object Reference Manual

Order Number: AA–MK72A–TE

This document contains reference information on all VAX LISP objects that are in the `VAX-LISP:` package but are not fully described in *Common LISP: The Language.*

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

digital equipment corporation
maynard, massachusetts

First Printing, July 1989

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

This document was prepared using VAX DOCUMENT, Version 1.1.

ML–S839

Contents

Index

Tables

Preface

This manual contains reference information on VAX LISP objects that are described in the *VAX LISP / VMS Program Development Guide*, *VAX LISP / VMS System Access Guide*, and *VAX LISP Implementation and Extensions to Common LISP* manuals, and that are contained in the VAX-LISP: package. Reference information for VAX LISP objects contained in the WINDOW-STREAM: package may be found in the *VAX LISP Implementation and Extensions to Common LISP* manual.

Intended Audience

This manual is intended for programmers with a good knowledge of both LISP and the programming interface to the VMS operating system.

Document Structure

This manual contains full descriptions of the functions, macros, variables, and constants in VAX LISP. Each function or macro description explains the function's or macro's use and shows its format, applicable arguments, return value, and examples of use. Each variable or constant description explains the variable's or constant's use and provides examples of its use. The descriptions are organized alphabetically.

Associated Documents

The following documents are relevant to VAX LISP/VMS programming:

- *VAX LISP / VMS Program Development Guide*
- *VAX LISP / VMS System Access Guide*
- *VAX LISP Implementation and Extensions to Common LISP*
- *VAX LISP / VMS System-Building Guide*
- *VAX LISP / VMS DECwindows Programming Guide*
- *VAX LISP Interface to VWS Graphics*
- *Common LISP: The Language*
- *VAX Architecture Handbook*
- *VMS DCL Dictionary*
- *VMS System Messages and Recovery Procedures Reference Manual*
- *VMS Record Management Services Manual*

- *VMS Utility Routines Manual*
- *VMS Command Definition Utility Manual*
- *VMS System Services Reference Manual*
- *VMS I/O User's Reference Manual: Part I*

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

Conventions

The following conventions are used in this manual:

Convention	**Meaning**
UPPERCASE	DCL commands and qualifiers and VMS file names are printed in uppercase characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: The examples directory (SYS$SYSROOT:[VAXLISP.EXAMPLES] by default) contains sample LISP source files.
`UPPERCASE TYPEWRITER`	Defined LISP functions, macros, variables, constants, and other symbol names are printed in uppercase `TYPEWRITER` characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: The `CALL-OUT` macro calls a defined external routine . . . .
`lowercase typewriter`	LISP forms are printed in the text in lowercase `typewriter` characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: `(setf example-1 (make-space))`
SANS SERIF	Format specifications of LISP functions and macros are printed in a sans serif typeface. For example: CALL-OUT *external-routine* &REST *routine-arguments*
italics	Lowercase *italics* in format specifications and in text indicate arguments that you supply; however, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters. For example: The *routine-arguments* must be compatible with the arguments defined in the call to the `DEFINE-EXTERNAL-ROUTINE` macro.
( )	Parentheses used in examples of LISP code and in format specifications indicate the beginning and end of a LISP form. For example: `(setq name lisp)`

Convention	Meaning
[ ]	Square brackets in format specifications enclose optional elements. For example: [*doc-string*] Square brackets do not indicate optional elements when they are used in the syntax of a directory name in a VMS file specification. Here, the square bracket characters must be included in the syntax. For example: `(pathname "MIAMI::DBA1:[SMITH]LOGIN.COM;4")`
{ }	In function and macro format specifications, braces enclose elements that are considered one unit of code. For example: {*keyword value*}
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered one unit of code, which can be repeated zero or more times. For example: {*keyword value*}*
&OPTIONAL	In function and macro format specifications, the word &OPTIONAL indicates that the arguments that follow it are optional. For example: PPRINT *object* &OPTIONAL *stream* Do not specify &OPTIONAL when you invoke a function or macro whose definition includes &OPTIONAL.
&REST	In function and macro format specifications, the word &REST indicates that an indefinite number of arguments may appear. For example: CALL-OUT *external-routine* &REST *routine-arguments* Do not specify &REST when you invoke a function or macro whose definition includes &REST.
&KEY	In function and macro format specifications, the word &KEY indicates that keyword arguments are accepted. For example: COMPILE-FILE *input-pathname* &KEY :LISTING :MACHINE-CODE :OPTIMIZE :OUTPUT-FILE :VERBOSE :WARNINGS Do not specify &KEY when you invoke a function or macro whose definition includes &KEY.
. . .	A horizontal ellipsis in a format specification means that the element preceding the ellipsis can be repeated. For example: *function-name . . .*
. . .	A vertical ellipsis in a code example indicates that all the information that the system would display in response to the function call is not shown; or, that all the information a user is to enter is not shown.

Convention	Meaning
Return	A word inside a box indicates that you press a key on the keyboard. For example: Return or Tab In code examples, carriage returns are implied at the end of each line. However, Return is used in some examples to emphasize carriage returns.
Ctrl/*x*	Two key names enclosed in a box indicate a control key sequence in which you hold down Ctrl while you press another key. For example: Ctrl/C or Ctrl/S
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1, then press and release another key.
mouse	The term *mouse* refers to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	By default, MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. You can rebind the mouse buttons.

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

ABORT Function

Unwinds the stack to the most recent CATCH-ABORT. The ABORT function is invoked whenever the cancel character (Ctrl/C) is typed at the keyboard. The VAX LISP top level uses the CATCH-ABORT macro, so that typing Ctrl/C puts you back at the top level.

Thus, you can use ABORT to cause an exit to the VAX LISP read-eval-print loop. In this way, you can partially simulate the action of the cancel character from within your code. (The cancel character also invokes the CLEAR-INPUT function on the *TERMINAL-IO* stream.)

NOTE

This function takes the place of a THROW to the CANCEL-CHARACTER-TAG tag in previous versions of VAX LISP.

Format

ABORT

Argument

None.

Return Value

Does not return.

Examples

1.
```
Lisp> (bind-keyboard-function #\^g
                              #'(lambda ()
                                  (clear-input *query-io*)
                                  (when (y-or-n-p "Are you sure? ")
                                    (abort)))
                              :level 5)

T
Lisp> (loop)
Ctrl/G
Are you sure? Y
Lisp>
```

- The call to the BIND-KEYBOARD-FUNCTION function binds a function to the key Ctrl/G.
- The user types Ctrl/G. (This is not echoed on the screen.)
- When the user types Y, the ABORT function returns control to the VAX LISP top level.

2.
```
Lisp>(setf *foo* nil)
NIL
Lisp>(defun foo ()
        (catch-abort (unwind-protect (unless *foo* (abort))
                                     (setf *foo* 3)))
        (+ *foo* 10))
FOO
Lisp>(foo)
13
```

The UNWIND-PROTECT cleanup form, (setf *foo* 3), is executed after ABORT is invoked.

ALIEN-DATA Function

Either dereferences a pointer to an alien structure's data vector or returns its address. For more information about alien structures, see Chapter 5 in the *VAX LISP/VMS System Access Guide*.

Format

ALIEN-DATA *alien-structure*

Argument

alien-structure
The alien structure for which you want to access the data vector.

Return Value

The data vector or an integer if the vector is in non-LISP space.

Examples

1.
```
Lisp> (setf y (make-text-string))
#<Alien Structure TEXT-STRING #<Vector #x21C511>>
Lisp> (setf (text-string-a y) "abc")
"abc"
Lisp> (alien-data y)
#(97 98 99 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
32 32)
Lisp> (text-string-a y)
"abc                     "
Lisp>
```

In this example, the alien structure is not constructed with the :DATA keyword. The accessor function is used to initialize the field. When ALIEN-DATA is used to access the data vector, it returns the unformatted contents of the vector. When the accessor function is used, it returns the field as a 24-byte string, according to the field description. This is the default.

2.

```
Lisp> (define-alien-structure text-string (a :string 0 24))
TEXT-STRING
Lisp> (setf x (make-text-string :data "this string is going to be very
long, longer than the 24 bytes allocated for the alien structure"))
#<Alien Structure TEXT-STRING this string is going to be very long,
longer than the 24 bytes allocated for the alien structure>
Lisp> (alien-data x)
"this string is going to be very long, longer than the 24 bytes
allocated for the alien structure"
Lisp> (text-string-a x)
"this string is going to "
Lisp>
```

This example defines an alien structure consisting of a single 24-byte text field. An instance of this structure is created with the :DATA keyword, making the pointer to the data vector point directly to a string rather than allocating a new vector. When you use ALIEN-DATA to access the data, it returns the entire string. When you use an accessor function to access the data, it returns only the first 24 bytes of the string according to the alien structure definition.

ALIEN-FIELD Function

Accesses the value of a field of a specified type from an alien structure. The function ignores the alien structure's predefined fields.

You can modify alien structures if you use the ALIEN-FIELD function with the SETF macro. This function is most useful for debugging a program that uses alien structures. The function can also be used to write your own accessor functions, for example, to access unnamed gaps in an alien structure.

For more information about alien structures, see Chapter 5 in the *VAX LISP / VMS System Access Guide*.

Format

ALIEN-FIELD *alien-structure field-type start end*

Arguments

alien-structure
The alien structure from which a field value is to be accessed.

field-type
The type of the field from which a value is to be accessed. It tells ALIEN-FIELD how to interpret the data. This argument can be either a keyword that names a built-in alien structure field type, a symbol (for a user-defined field type), or a list whose first element names the field type.

start
A rational number that specifies the start position (in 8-bit bytes) of a field in the alien structure's data area. This value is inclusive and zero-based.

end
A rational number that specifies the end position (in 8-bit bytes) of a field in the alien structure's data area. This value is exclusive.

Return Value

The value of the specified alien structure field.

Example

```
Lisp> (define-alien-structure space
        (area-1 :unsigned-integer 0 4 :default 22)
        (area-2 :unsigned-integer 4 8 :default 2764))
SPACE
Lisp> (setf space-record (make-space))
#<Alien Structure SPACE #x45FA60>
Lisp> (space-area-1 space-record)
22
Lisp> (space-area-2 space-record)
2764
Lisp> (alien-field space-record :unsigned-integer 0 4)
22
Lisp> (alien-field space-record :unsigned-integer 4 8)
2764
Lisp> (alien-field space-record :unsigned-integer 0 8)
11871289606166
```

This example illustrates:

- If you specify the ALIEN-FIELD function with the same field types and positions that are in the definition of an alien structure, the data you access is the same as if you had accessed it with that structure's default accessor functions.
- If you specify the ALIEN-FIELD function with field types and positions different from those in a defined alien structure, the interpretation of the data you access could be different, depending on the field type and field positions you specify.

ALIEN-STRUCTURE-LENGTH Function

Returns the length of an alien structure in bytes.

Format

ALIEN-STRUCTURE-LENGTH *alien-structure*

Argument

alien-structure
The alien structure whose length is to be returned.

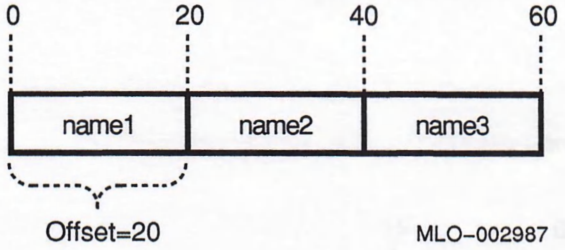
Return Value

The length of the alien structure in bytes.

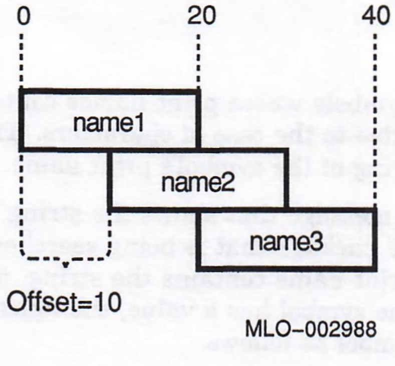
Examples

The following examples illustrate the use of the ALIEN-STRUCTURE-LENGTH function. The diagram after each example illustrates why it returns a specific value.

1.
```
Lisp> (define-alien-structure example1
         (name :string 0 20 :occurs 3 :offset 20))
EXAMPLE1
Lisp> (alien-structure-length (make-example1))
60
```

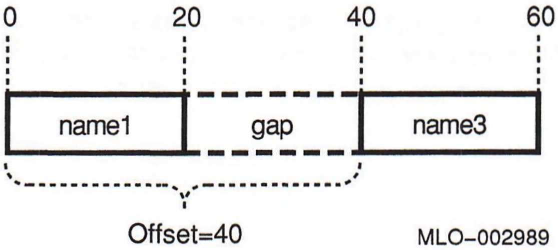


2.
```
Lisp> (define-alien-structure example2
         (name :string 0 20 :occurs 3 :offset 10))
EXAMPLE2
Lisp> (alien-structure-length (make-example2))
40
```



In EXAMPLE2, the offset overlaps so that the last part of the information stored in NAME1 becomes the first part of the information stored in NAME2, and so on.

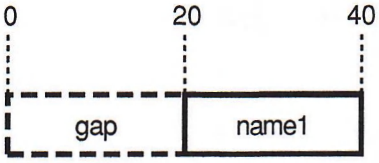
3.
```
Lisp> (define-alien-structure example3
        (name :string 0 20 :occurs 2 :offset 40))
EXAMPLE3
Lisp> (alien-structure-length (make-example3))
60
```



MLO-002989

In EXAMPLE3 and EXAMPLE4, the gaps are counted as part of the length of the structure.

4.
```
Lisp> (define-alien-structure example4 (name :string 20 40))
EXAMPLE4
Lisp> (alien-structure-length (make-example4))
40
```



MLO-002990

APROPOS Function

Searches through packages for symbols whose print names contain a specified string. The function is not sensitive to the case of characters. The string can be either the print name or a substring of the symbol's print name.

The APROPOS function displays a message that shows the string that is being searched for and the name of the package that is being searched. When the function finds a symbol whose print name contains the string, the function displays the symbol's name. If the symbol has a value, the function displays the phrase "has a value" after the symbol as follows:

```
*MY-SYMBOL*, has a value
```

If the symbol has a function definition, the function displays the phrase "has a definition" after the symbol as follows:

```
MY-FUNCTION, has a definition
```

APROPOS Function

In VAX LISP, the APROPOS function uses the DO-SYMBOLS macro rather than the DO-ALL-SYMBOLS macro. As a result, the function displays by default only symbols that are accessible from the current or specified package. For information on packages, see *Common LISP: The Language.*

Format

APROPOS *string* **&OPTIONAL** *package*

Arguments

string
The string to be sought in the symbols' print names. If you specify a symbol for this argument, the symbol's print name is used.

package
An optional argument. If you specify the argument, the symbols in the specified package are searched. If you specify T, all packages are searched. If you do not specify the argument, the symbols that are accessible in the current package are searched.

Return Value

No value.

Example

```
Lisp> (apropos "*print")

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

Searches the package USER for the string "*PRINT" and displays a list of the symbols that contain the specified string.

APROPOS-LIST Function

Searches through packages for symbols whose print names contain a specified string. The function is not sensitive to the case of characters. The string can be either the print name or a substring of the symbol's print name.

When the function completes its search, it returns a list of the symbols whose print names contain the string.

In VAX LISP, the APROPOS-LIST function uses the DO-SYMBOLS macro rather than the DO-ALL-SYMBOLS macro. As a result, the function includes by default only symbols that are accessible from the current package in the list it returns. For information on packages, see *Common LISP: The Language.*

Format

APROPOS-LIST *string* **&OPTIONAL** *package*

Arguments

string
The string to be sought in the symbols' print names. If you specify a symbol for this argument, the symbol's print name is used.

package
An optional argument. If you specify the argument, the symbols in the specified package are searched. If you specify T, all packages are searched. If you do not specify the argument, the symbols that are accessible in the current package are searched.

Return Value

A list of the symbols whose print names contain the string.

Example

```
Lisp> (apropos-list "array")
(ARRAY-TOTAL-SIZE ARRAY-DIMENSION ARRAY-DIMENSIONS
SIMPLE-ARRAY ARRAY-DIMENSION-LIMIT ARRAY-ELEMENT-TYPE
ARRAYP *PRINT-ARRAY* ARRAY-RANK ARRAY-RANK-LIMIT
MAKE-ARRAY ARRAY-TOTAL-SIZE-LIMIT ARRAY-ROW-MAJOR-INDEX
ADJUST-ARRAY ARRAY ARRAY-IN-BOUNDS-P ADJUSTABLE-ARRAY-P
ARRAY-HAS-FILL-POINTER-P)
```

Searches the symbols that are accessible in the current package for the string "ARRAY" and returns a list of the symbols that contain the specified string.

AREA-SEGMENT-LIMIT Function

Returns the maximum number of segments that the specified area may occupy before being garbage collected. See the *VAX LISP Implementation and Extensions to Common LISP* manual for details on garbage collection in VAX LISP.

This function may be used with the `SETF` macro, but only on the ephemeral areas (the *area* argument is 0, 1, or 2).

Format

AREA-SEGMENT-LIMIT &OPTIONAL *area*

Argument

area

A keyword or number indicating the area of memory:

`:DYNAMIC`	Total dynamic space, including ephemeral areas. This is the default.
`0`	The most transient ephemeral area, E0.
`1`	The second ephemeral area, E1.
`2`	The least transient ephemeral area, E2.

Values of `T`, `NIL`, or `:DEFAULT` are interpreted as `:DYNAMIC`.

Return Value

An integer.

Example

```
Lisp> (area-segment-limit 0)
10
Lisp> (area-segment-limit t)
22
```

AREA-SEGMENTS Function

Returns the number of segments that are currently allocated to the specified area of dynamic memory. See the *VAX LISP Implementation and Extensions to Common LISP* manual for details on VAX LISP memory management and the garbage collector.

AREA-SEGMENTS Function

Format

AREA-SEGMENTS *area*

Argument

area

A keyword or number indicating the area of memory:

`:DYNAMIC`	Total dynamic space, excluding ephemeral areas. This is the default.
`:READ-ONLY`	Read-only space contains much of the LISP system and is never garbage collected.
`:STACK`	Stack space contains several kinds of stacks, and is scanned as part of garbage collection.
`:STATIC`	Static space contains user-allocated objects that are guaranteed not to be moved by the garbage collector.
0	The most transient ephemeral area, E0.
1	The second ephemeral area, E1.
2	The least transient ephemeral area, E2.

Values of `T`, `NIL`, or `:DEFAULT` are interpreted as `:DYNAMIC`.

Return Value

An integer.

Example

```
Lisp> (area-segments :read-only)
79
Lisp> (area-segments t)
4
```

ATTACH Function

Connects your terminal to a process and puts the current LISP process into a VMS hibernation state, a state in which a process is inactive but can become active at a later time. You can use this function to switch terminal control from one process to another.

The ATTACH function is similar to the DCL ATTACH command. For information on the ATTACH command, see the *VMS DCL Dictionary*.

NOTES

The ATTACH function can be used only if LISP is invoked from DCL; it cannot be used if LISP is invoked from another Command Language Interpreter (CLI).

Be careful using this function under DECwindows, both in the development environment and in your programs. Since ATTACH causes LISP to hibernate, no events can be processed. If events are queued and LISP does not respond within a server-defined timeout, the X server aborts the connection to the LISP process.

Format

ATTACH *process*

Argument

process

The name or identification (PID) of the process to which your terminal is to be connected. To specify the process name, use a string or a symbol; to specify the PID, use an integer.

Return Value

Unspecified.

Examples

1.

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

- The call to the SPAWN function creates a subprocess named SMITH_1.
- The DCL ATTACH command attaches your terminal back to the process SMITH.
- The call to the VAX LISP ATTACH function returns control to the process SMITH_1.

2.

```
Lisp> (defun attach-main nil
        (attach (second (get-process-information
                         nil
                         :owner-pid))))
ATTACH-MAIN
```

Defines a function that attaches back to the main process if the LISP system is running as a subprocess.

BIND-KEYBOARD-FUNCTION Function

Binds an ASCII keyboard control character (characters of codes 0 to 31) to a function. When a control character is bound to a function, you can execute the function by typing the control character on your terminal keyboard. A function bound in this way is called a keyboard function.

When you type the control character, the LISP system is interrupted at its current point, and the function the control character is bound to is called asynchronously. The LISP system then evaluates the function and returns control to where the interruption occurred.

You can delete the binding of a function and a control character by using the UNBIND-KEYBOARD-FUNCTION function. You can use the GET-KEYBOARD-FUNCTION function to get information about a function that is bound to a control character.

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

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

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

NOTE

> When you bind a control character to a function, the stream bound to the *TERMINAL-IO* variable must be connected to your terminal.

See *VAX LISP Implementation and Extensions to Common LISP* for an explanation of calling functions asynchronously.

Format

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

Arguments

control-character
The ASCII control character to be bound to the function. You can bind a function to any control character except Ctrl/Q or Ctrl/S.

function
The function to which the control character is to be bound.

:ARGUMENTS
A list containing arguments to be passed to the specified function when it is called. The arguments in the list are evaluated when the BIND-KEYBOARD-FUNCTION function is called.

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

Return Value

T.

Examples

```
Lisp> (bind-keyboard-function #\^B #'break)
T
Lisp> Ctrl/B
Break>
```

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

2.
```
Lisp> (bind-keyboard-function #\^E #'ed)
T
Lisp> Ctrl/E
 .
 .    (now in the Editor)
 .
```

Binds Ctrl/E to the ED function. You can then invoke the Editor by typing Ctrl/E.

BREAK Function

Invokes a break loop. A break loop is a nested read-eval-print loop. For more information about break loops, see Chapter 4 of the *VAX LISP/VMS Program Development Guide.*

Format

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

Arguments

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

args
The arguments that are passed to the FORMAT function as arguments for the format string.

Return Value

When the CONTINUE function is called to exit the break loop, the BREAK function returns NIL; otherwise, no value is returned.

Example

```
(when (unusual-situation-p status)
    (break "Unusual situation ~D encountered. Please investigate"
      status))
```

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

CALL-BACK-ROUTINE Type Specifier

An alien structure that can be passed to an external routine during callout. See Chapter 4 in the *VAX LISP/VMS System Access Guide* for more information about the callback facility.

Format

CALL-BACK-ROUTINE

CALL-OUT Macro

Calls a defined external routine. If you specify an external routine that has not been defined with the DEFINE-EXTERNAL-ROUTINE macro, the LISP system signals an error.

For information about how to use the VAX LISP callout facility, see Chapter 4 in *VAX LISP/VMS System Access Guide.*

Format

CALL-OUT *external-routine* **&REST** *args*

Arguments

external-routine
The name of a defined external routine.

args
Arguments to be passed to the external routine. The arguments correspond by position to the arguments defined for the routine. The LISP system evaluates the argument expressions before the external routine is called. You can omit an optional argument by specifying an expression whose value is `NIL`. The corresponding position in the argument list will contain a 0 to coincide with the VAX Procedure Calling Standard. If you specify fewer arguments than were specified in the definition, the argument list will contain only the number of arguments actually supplied. LISP signals an error if you supply more arguments than were specified in the definition.

Return Value

If the `:RESULT` option of the `DEFINE-EXTERNAL-ROUTINE` macro was specified, the external routine's result is returned; otherwise, no value is returned.

Example

```
Lisp> (define-external-routine (smg$create_pasteboard
                                  :file "SMGSHR"
                                  :result integer)
        (new-pasteboard-id :lisp-type integer
                           :vax-type :unsigned-longword
                           :access :in-out)
        (output-device :lisp-type string)
        (pb-rows :lisp-type integer :access :in-out)
        (pb-columns :lisp-type integer :access :in-out)
        (preserve-screen-flag :lisp-type integer
                              :vax-type :unsigned-longword))
SMG$CREATE_PASTEBOARD
Lisp> (defvar *pasteboard-id* -1)
*PASTEBOARD-ID*
Lisp> (call-out smg$create_pasteboard *pasteboard-id*
                                      nil nil nil 1)
1
```

- The call to the `DEFINE-EXTERNAL-ROUTINE` macro defines the VMS Screen Management Facility (SMG$) routine called `SMG$CREATE_PASTEBOARD`.
- The call to the `DEFVAR` macro defines a special variable which contains the pasteboard ID returned by the external routine.

- The call to the CALL-OUT macro invokes the external routine SMG$CREATE_PASTEBOARD, specifying the special variable to receive the pasteboard ID. Three arguments are omitted, and a preserve-screen-flag of 1 is given. The result status is returned.

CATCH-ABORT Macro

Catches the throw to the VAX LISP top level generated by the ABORT function. (For example, the ABORT function is invoked when the cancel character (Ctrl/C) is typed at the keyboard.) Thus, you can use CATCH-ABORT to alter the behavior whenever ABORT is called.

NOTE

> This macro takes the place of `(catch 'cancel-character-tag (...))` forms in previous versions of VAX LISP.

Format

CATCH-ABORT {*body*}*

Argument

body
One or more LISP forms.

Return Value

Unspecified.

Example

```
Lisp> (defun trapper ()
        (catch-abort (loop))
        (princ "Execution came through here"))
TRAPPER
Lisp> (TRAPPER)
Ctrl/C
Execution came through here
"Execution came through here"
Lisp>
```

- The TRAPPER function sets up a catcher for the cancel character, then enters an infinite loop.
- The user types Ctrl/C.
- The PRINC function prints a string, indicating that execution continued after the CATCH-ABORT form rather than returning directly to the Lisp> prompt.

CHAR-NAME-TABLE Function

Displays a formatted list of the VAX LISP character names.

Format

CHAR-NAME-TABLE

Argument

None.

Return Value

No value.

Example

```
Lisp> (char-name-table)

Hex Code  Preferred Name  Other Names

   00     NULL            NUL
   01     ^A              SOH
   02     ^B              STX
   03     ^C              ETX
   04     ^D              EOT
   05     ^E              ENQ
   06     ^F              ACK
   07     BELL            ^G  BEL
   08     BACKSPACE       ^H  BS
   09     TAB             ^I  HT
   0A     LINEFEED        ^J  LF
   0B     ^K              VT
   0C     PAGE            ^L  FORMFEED  FF
   0D     RETURN          ^M  CR
   0E     ^N              SO
   0F     ^O              SI
   10     ^P              DLE
   11     ^Q              XON  DC1
   12     ^R              DC2
   13     ^S              XOFF  DC3
   14     ^T              DC4
   15     ^U              NAK
   16     ^V              SYN
   17     ^W              ETB
   18     ^X              CAN
   19     ^Y              EM
   1A     ^Z              SUB
   1B     ESCAPE          ESC  ALTMODE
   1C     FS
   1D     GS
   1E     RS
   1F     US
   20     SPACE           SP
```

7F	RUBOUT	DELETE	DEL
84	IND		
85	NEL		
86	SSA		
87	ESA		
88	HTS		
89	HTJ		
8A	VTS		
8B	PLD		
8C	PLU		
8D	RI		
8E	SS2		
8F	SS3		
90	DCS		
91	PU1		
92	PU2		
93	STS		
94	CCH		
95	MW		
96	SPA		
97	EPA		
9B	CSI		
9C	ST		
9D	OSC		
9E	PM		
9F	APC		
FF	NEWLINE		

COMMAND-LINE-ENTITY-P Function

Returns two values indicating the presence or absence of the specified entity on the command line that invoked LISP. This function provides an interface to the CLI$PRESENT routine described in the *VMS Utility Routines Manual*. Refer to that manual and to the *VMS Command Definition Utility Manual* for a description of defining DCL commands and obtaining values from the command line in your program.

The COMMAND-LINE-ENTITY-P function is especially useful in a user-built LISP system that is invoked by a defined DCL command. See *VAX LISP/VMS System-Building Guide* for information on user-built LISP systems.

Format

COMMAND-LINE-ENTITY-P *entity-desc*

Argument

entity-desc

A character string or symbol. If you supply a symbol, the print name of the symbol is used. See the description of the *entity-desc* argument to CLI$PRESENT in the *VMS Utility Routines Manual* for information about the meaning of this argument.

Return Values

Two values. The meaning of these values is as follows:

First Value	Second Value	Meaning
T	T	Entity was explicitly specified as present.
T	NIL	Entity was present by default.
NIL	T	Entity was explicitly negated with NO.
NIL	NIL	Entity was absent by default.

COMMAND-LINE-ENTITY-VALUE Function

Returns the value of the specified entity on the command line that invoked LISP. This function provides an interface to the CLI$GET_VALUE routine described in the *VMS Utility Routines Manual.* Refer to that manual and to the *VMS Command Definition Utility Manual* for a description of defining DCL commands and obtaining values from the command line in your program.

The COMMAND-LINE-ENTITY-VALUE function is especially useful in a user-built LISP system that is invoked by a defined DCL command. See the *VAX LISP/VMS System-Building Guide* for information on user-built LISP systems.

Format

COMMAND-LINE-ENTITY-VALUE *entity-desc*

Argument

entity-desc

A character string or symbol. If you supply a symbol, the print name of the symbol is used. See the description of the **entity-desc** argument to CLI$GET_VALUE in the *VMS Utility Routines Manual* for information about the meaning of this argument.

Return Values

Two values:

1. A string containing the first or next value of the specified entity, depending on whether this is the first request for this entity or a subsequent request. If the entity is not present, has no value, or if all values for this entity have been obtained, NIL is returned.

2. A character that is the character delimiter that preceded the returned entity. This is normally a comma (#\,) but may be a plus sign (#\+) to indicate concatenation.

COMMON-AST-ADDRESS Parameter

Specifies the address of a routine, supplied by VAX LISP/VMS, that initially handles all ASTs. This parameter must be given as the *astadr* argument to all external routines that can cause an AST. No other object can be passed as the *astadr* argument. Use the `:VALUE` mechanism to pass this parameter.

Format

COMMON-AST-ADDRESS

Example

See the description of `INSTATE-INTERRUPT-FUNCTION` for an example of the use of `COMMON-AST-ADDRESS`.

COMPILEDP Function

A predicate that checks whether an object is a symbol that has a compiled function definition.

Format

COMPILEDP *name*

Argument

name
The symbol whose function or macro definition is to be checked.

Return Value

Returns the interpreted function or macro definition, if the symbol has an interpreted definition that was compiled with the `COMPILE` function. Returns `T`, if the symbol has a compiled definition that was not compiled with the `COMPILE` function. Returns `NIL`, if the symbol does not have a compiled function definition.

Example

```
Lisp> (defun add2 (x) (+ x 2))
ADD2
Lisp> (compiledp 'add2)
NIL
Lisp> (compile 'add2)
ADD2
ADD2
Lisp> (compiledp 'add2)
(LAMBDA (X) (BLOCK ADD2 (+ X 2)))
```

- The call to the DEFUN macro defines a function named ADD2.
- The first call to the COMPILEDP function returns NIL, because the function ADD2 has not been compiled.
- The call to the COMPILE function compiles the function ADD2.
- The second call to the COMPILEDP function returns the interpreted function definition, because the function ADD2 was previously compiled.

COMPILE-FILE Function

Compiles a specified LISP source file and writes the compiled code as a binary fast-loading file (file type .FAS).

Format

COMPILE-FILE *input-pathname* **&KEY :LISTING :MACHINE-CODE :OPTIMIZE :OUTPUT-FILE :VERBOSE :WARNINGS**

Arguments

input-pathname

A pathname, namestring, symbol, or stream. The VAX LISP Compiler uses the value of the *DEFAULT-PATHNAME-DEFAULTS* variable to fill in file specification components that are not included in your *input-pathname*. The file type defaults to .LSP.

:LISTING

Specifies whether the Compiler is to produce a listing file. The value can be T, NIL, or a pathname, namestring, symbol, or stream. If you specify T, the Compiler produces a listing file. The listing file is assigned the same name as the source file with the file type .LIS, and is placed in the directory that contains the source file.

If you specify NIL, no listing is produced. The default value is NIL.

If you specify a pathname, namestring, symbol, or stream, the Compiler uses the value as the specification of the listing file. The Compiler uses the .LIS file type and the value of the *input-pathname* to fill the components of the file specification that are not included in your pathname, namestring, symbol, or stream.

:MACHINE-CODE

Specifies whether the Compiler is to include the machine code that it produces for each function and macro that it compiles in the listing file. The value can be T or NIL. If you specify T, the listing file contains the machine code. If you specify NIL, the listing file does not contain the machine code. The default value is NIL.

:OPTIMIZE

Specifies the optimization qualities that the Compiler is to use during compilation. The value must be a list of sublists. Each sublist must contain a symbol and a value, which specify the optimization qualities and corresponding values that the Compiler is to use during compilation. For example:

```
'((space 2) (safety 1))
```

The default value for each quality is one. For a detailed discussion of Compiler optimizations, see Chapter 2 of the *VAX LISP/VMS Program Development Guide*.

:OUTPUT-FILE

Specifies whether the Compiler is to produce a fast-loading file. The value can be T, NIL, or a pathname, namestring, symbol, or stream. If you specify T, the Compiler produces a fast-loading file. The output file is assigned the same name as the source file with the file type .FAS and is placed in the directory that contains the source file. The default value is T.

If you specify NIL, no fast-loading file is produced.

If you specify a pathname, namestring, symbol, or stream, the Compiler uses that value as the specification of the output file. The Compiler uses the .FAS file type and the value of the *input-pathname* to fill the components of the file specification that are not included in your pathname, namestring, symbol, or stream.

:VERBOSE

Specifies whether the Compiler is to display the name of functions and macros that it compiles. The value can be T or NIL. If you specify T, the Compiler displays the name of each function and macro. If a listing file exists, the Compiler also includes the names in the listing file. If you specify NIL, the names are not displayed or included in the listing file. The default value is the value of the *COMPILE-VERBOSE* variable (by default, T).

:WARNINGS

Specifies whether the Compiler is to display warning messages. The value can be T or NIL. If you specify T, the Compiler displays warning messages. If a listing file exists, the Compiler also includes the messages in the listing file. If you specify NIL, warning messages are not displayed or included in the listing file. The default value is the value of the *COMPILE-WARNINGS* variable (by default, T).

Return Value

If the Compiler generated an output file, a namestring is returned; otherwise, NIL is returned.

Examples

1.
```
Lisp> (compile-file "FACTORIAL" :verbose t)

Starting compilation of file DBA1:[SMITH]FACTORIAL.LSP;1

FACTORIAL compiled.

Finished compilation of file DBA1:[SMITH]FACTORIAL.LSP;1
0 Errors, 0 Warnings
"DBA1:[SMITH]FACTORIAL.FAS;1"
```

Compiles the file FACTORIAL.LSP, which is in the current directory. A fast-loading file named FACTORIAL.FAS is produced. The compilation is logged to the terminal, because the :VERBOSE keyword is specified with the value T.

2.
```
Lisp> (compile-file "FACTORIAL" :output-file nil
                                :listing t
                                :warnings nil
                                :verbose nil)
NIL
```

Compiles the file FACTORIAL.LSP, which is in the current directory. A fast-loading file is not produced, because the :OUTPUT-FILE keyword is specified with the value NIL. A listing file named FACTORIAL.LIS is produced. Warning messages are suppressed, because the :WARNINGS keyword is specified with the value NIL.

COMPILE-VERBOSE Variable

Controls the amount of information that the Compiler displays.

The COMPILE-FILE function binds the *COMPILE-VERBOSE* variable to the value supplied by the :VERBOSE keyword. If the :VERBOSE keyword is not specified, the function uses the existing value of the *COMPILE-VERBOSE* variable. If the value is not NIL, the Compiler displays the name of each function as it is compiled; if the value is NIL, the Compiler does not display the function names. The default value is T.

Example

```
Lisp> (compile-file 'math)
Starting compilation of file DBA1:[SMITH]MATH.LSP;1

FACTORIAL compiled.
FIBONACCI compiled.
```

```
Finished compilation of file DBA1:[SMITH]MATH.LSP;1
0 Errors, 0 Warnings
"DBA1:[SMITH]MATH.FAS;1"
Lisp> (SETF *COMPILE-VERBOSE* NIL)
NIL
Lisp> (compile-file 'math)
"DBA1:[SMITH]MATH.FAS;2"
```

- The first call to the COMPILE-FILE function shows the output the Compiler displays during the compilation of a file, when the *COMPILE-VERBOSE* variable is the default, T.
- The call to the SETF macro sets the value of the variable to NIL.
- The second call to the COMPILE-FILE function compiles the file without displaying output, because the variable's value is NIL.

COMPILE-WARNINGS Variable

Controls whether the Compiler displays warning messages during a compilation.

The COMPILE-FILE function binds the *COMPILE-WARNINGS* variable to the value supplied with the :WARNINGS keyword. If the :WARNINGS keyword is not specified, the function uses the existing value of the *COMPILE-WARNINGS* variable. If the value is not NIL, the Compiler displays warning messages; if the value is NIL, the Compiler does not display warning messages. The default value is T.

NOTE

The Compiler always displays fatal and continuable error messages.

Example

```
Lisp> (compile-file 'math)
Starting compilation of file DBA1:[SMITH]MATH.LSP;2

Warning in FACTORIAL
  N bound but value not used.
FACTORIAL compiled.
Warning in FIBONACCI
  N bound but value not used.
FIBONACCI compiled.

Finished compilation of file DBA1:[SMITH]MATH.LSP;2
0 Errors, 2 Warnings
"DBA1:[SMITH]MATH.FAS;3"
Lisp> (setf *compile-warnings* nil)
NIL
Lisp> (compile-file 'math)
Starting compilation of file DBA1:[SMITH]MATH.LSP;2

FACTORIAL compiled.
FIBONACCI compiled.

Finished compilation of file DBA1:[SMITH]MATH.LSP;2
0 Errors, 2 Warnings
"DBA1:[SMITH]MATH.FAS;4"
```

- The first call to the COMPILE-FILE function shows the output that the Compiler displays during the compilation of a file, when the *COMPILE-WARNINGS* variable is the default, T.
- The call to the SETF macro sets the value of the variable to NIL.
- The second call to the COMPILE-FILE function compiles the file without displaying warning messages in the output, because the variable's value is NIL.

CONTINUE Function

Enables you to exit the break loop. When you call this function, it causes the BREAK function to return NIL and the evaluation of your program to continue from the point at which the break loop was entered.

Format

CONTINUE

Argument

None.

Return Value

NIL.

Example

```
Lisp> (bind-keyboard-function #\^B #'break)
Lisp> (load "FILEB.LSP")
; Loading contents of file LISPW$:[SMI...
^B
Break> (load "FILEA.LSP")
; Loading contents of file LISPW$:[SMITH]FILEA.LSP;1
;   FUNCTION-A
; Finished loading LISPW$:[SMITH]FILEA.LSP;1
T
Break> (continue)
Continuing from break loop...
;   FUNCTION-B
; Finished loading LISPW$:[SMITH]FILEB.LSP;1
T
Lisp>
```

- The BREAK function is bound to Ctrl/B.
- FILEB.LSP is loaded.

- Realizing that FILEA.LSP, which is needed to initialize an environment for FILEB.LSP, is not yet loaded, the programmer invokes the BREAK loop.
- FILEA.LSP is then loaded.
- Finally, the call to the CONTINUE function continues the loading of FILEB.LSP and then returns the programmer to the top-level loop.

CRITICAL-SECTION Macro

Executes the forms in its body as a "critical section." During the execution of a critical section, all interrupt functions are blocked and queued for later execution. Ctrl/C is also blocked, so a critical section must neither loop nor cause errors. It is an error to perform I/O or to call the WAIT function in a critical section.

If an error occurs during a critical section, VAX LISP invokes the Debugger, and temporarily removes the restrictions on interrupts so you can type to the Debugger. If you continue from the Debugger, LISP restores the restrictions on interrupts before continuing. However, LISP is open to interruptions while you are debugging the code.

Format

CRITICAL-SECTION {*form*}*

Argument

form
Form(s) to be executed as a critical section.

Return Value

Value(s) of the last form that was executed.

Example

```
Lisp> (defun restore-to-free-list (cons-cell)
        (critical-section
          (setf (cdr cons-cell) *head-of-free-list*
                *head-of-free-list* cons-cell)))
RESTORE-TO-FREE-LIST
```

This example defines a function that restores a cons cell to the head of a list of free cells. During the call to SETF, the list is in an inconsistent state, because the special variable *HEAD-OF-FREE-LIST* does not point to the head of the list. An interrupting function that used *HEAD-OF-FREE-LIST* to remove an element from the list would break the list. Therefore, RESTORE-TO-FREE-LIST uses the CRITICAL-SECTION macro to ensure that the SETF call completes without interruption.

DEBUG Function

Invokes the VAX LISP Debugger. For information on how to use the VAX LISP Debugger, see Chapter 4 of the *VAX LISP/VMS Program Development Guide.*

Format

DEBUG

Argument

None.

Return Value

Returns NIL. You can cause the Debugger to return other values. See Chapter 4 of the *VAX LISP/VMS Program Development Guide.*

Example

```
Lisp> (debug)
Control Stack Debugger
Apply #5: (DEBUG)
Debug 1>
```

Invokes the VAX LISP Debugger. When you invoke the Debugger, it displays an identifying message, stack frame information, and the Debugger prompt.

DEBUG-CALL Function

Returns a list representing the current debug frame function call. This function is a debugging tool and takes no arguments. The list returned by the DEBUG-CALL function can be used to access the values passed to the function in the current stack frame.

Format

DEBUG-CALL

Argument

None.

Return Value

A list representing the current debug frame function call. NIL is returned if this function is called outside the Debugger.

Example

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

Fatal error in function SCHAR (signaled with ERROR).
Argument must be a simple-string: "          "

Control Stack Debugger
Apply #4: (SCHAR "          " 3)
Debug 1> (type-of (second (debug-call)))

(STRING 10)
Debug 1> ret #\space
#\SPACE
```

In this case, the function in the current stack frame is SCHAR. The call to (DEBUG-CALL) returns the list (SCHAR "   " 3). The form (second (debug-call)) returns the first argument to SCHAR in the current stack frame. Calling TYPE-OF with this LISP object determines that the first argument to SCHAR is of type (STRING 10) and not a simple string. See the TRACE macro description for another example of the use of the DEBUG-CALL function.

DEBUG-PRINT-LENGTH Variable

Controls the output that the VAX LISP Debugger, Stepper, and Tracer facilities display. This variable controls the number of objects these facilities can display at each level of a nested data object. The variable's value can be either a positive integer or NIL. If the value is a positive integer, the integer indicates the number of objects at each level of a nested object to be displayed. If the value is NIL, no limit is on the number of objects that can be displayed. The default value is NIL.

The value of this variable might cause the printer to truncate output. An ellipsis (. . .) indicates truncation.

This variable is similar to the *PRINT-LENGTH* variable described in *Common LISP: The Language*.

Example

```
Lisp> (setf alphabet '(a b c d e f g h i j k))
(A B C D E F G H I J K)
Lisp> (setf *debug-print-length* 5)
5
Lisp> (+ 2 ALPHABET)

Fatal error in function + (signaled with ERROR).
Argument must be a number: (A B C D E F G H I J K)

Control Stack Debugger

Apply #5: (+ 2 (A B C D E ...))

Debug 1> (SETF *DEBUG-PRINT-LENGTH* 3)
3
Debug 1> WHERE

Apply #5: (+ 2 (A B C ...))
```

- The call to the SETF macro sets the symbol ALPHABET to a list of single-letter symbols.
- The value of the *DEBUG-PRINT-LENGTH* variable is set to 5.
- The illegal call to the plus sign (+) function causes the LISP system to invoke the Debugger. The Debugger displays only five elements of the list that is the value of the symbol ALPHABET the first time it displays the stack frame numbered 5.
- The call to the SETF macro within the Debugger sets the value of the *DEBUG-PRINT-LENGTH* variable to 3.
- The Debugger displays three elements of the list, after you change the value of the variable.

DEBUG-PRINT-LEVEL Variable

Controls the output that the VAX LISP Debugger, Stepper, and Tracer facilities display. This variable controls the number of levels of a nested object these facilities can display. The variable's value can be either a positive integer or NIL. If the value is a positive integer, the integer indicates the number of levels of a nested object to be displayed. If the value is NIL, no limit is on the number of levels that can be displayed. The default value is NIL.

The value of this variable might cause the printer to truncate output. A number sign (#) indicates truncation.

This variable is similar to the *PRINT-LEVEL* variable described in *Common LISP: The Language*.

DEBUG-PRINT-LEVEL Variable

Example

```
Lisp> (setf alphabet '(a (b (c (d (e))))))
(A (B (C (D (E)))))
Lisp> (setf *debug-print-level* 3)
3
Lisp> (+ 2 ALPHABET)

Fatal error in function + (signaled with ERROR).
Argument must be a number: (A (B (C (D (E)))))

Control Stack Debugger

Apply #5: (+ 2 (A (B #)))

Debug 1> (setf *debug-print-level* nil)
NIL
Debug 1> where

Apply #5: (+ 2 (A (B (C (D (E))))))
```

- The call to the SETF macro sets the symbol ALPHABET to a nested list.
- The value of the *DEBUG-PRINT-LEVEL* variable is set to 3.
- The illegal call to the plus sign (+) function causes the LISP system to invoke the Debugger. The Debugger displays only three levels of the nested list (that is the value of the symbol ALPHABET) the first time it displays the stack frame numbered 5.
- The call to the SETF macro within the Debugger sets the value of the *DEBUG-PRINT-LEVEL* variable to NIL.
- The Debugger displays all the levels of the nested list, after you change the value of the variable.

DEFAULT-DIRECTORY Function

Returns a pathname with the host, device, and directory fields filled with the values of the current default directory.

The DEFAULT-DIRECTORY function is similar to the DCL SHOW DEFAULT command. For information about the SHOW DEFAULT command, see the *VMS DCL Dictionary*.

You can change the default directory by using the SETF macro. Setting your default directory with this macro also resets the value of the *DEFAULT-PATHNAME-DEFAULTS* variable. Performing this operation is similar to using the DCL SET DEFAULT command. See *VAX LISP Implementation and Extensions to Common LISP* and *Common LISP: The Language* for more information on pathnames and the *DEFAULT-PATHNAME-DEFAULTS* variable.

Note that the directory must exist for the change of directory to succeed.

DEFAULT-DIRECTORY Function

Format

DEFAULT-DIRECTORY

Argument

None.

Return Value

The pathname that refers to the default directory.

Examples

1.

```
Lisp> (default-directory)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
:DIRECTORY "SMITH" :NAME NIL :TYPE NIL
:VERSION NIL)
Lisp> (setf (default-directory) "[.tests]")
"[.TESTS]"
Lisp> (default-directory)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
:DIRECTORY "SMITH.TESTS" :NAME NIL :TYPE NIL
:VERSION NIL)
```

- The first call to the DEFAULT-DIRECTORY function returns the pathname that points to the default directory.
- The call to the SETF macro changes the value of the default directory to SMITH.TESTS.
- The second call to the DEFAULT-DIRECTORY function verifies the directory change.

2.

```
Lisp> (default-directory)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
:DIRECTORY "SMITH.TESTS" :NAME NIL :TYPE NIL
:VERSION NIL)
Lisp> *default-pathname-defaults*
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
:DIRECTORY "SMITH.TESTS" :NAME NIL :TYPE NIL
:VERSION NIL)
Lisp> (namestring (default-directory))
"DBA1:[SMITH.TESTS]"
Lisp> (setf (default-directory) "[-]")
"[-]"
Lisp> (namestring (default-directory))
"DBA1:[SMITH]"
Lisp> (namestring *default-pathname-defaults*)
"DBA1:[SMITH]"
```

- The first call to the DEFAULT-DIRECTORY function returns the pathname that points to the default directory.
- The call to the *DEFAULT-PATHNAME-DEFAULTS* variable shows that its value is the same as the value returned by the DEFAULT-DIRECTORY function.
- The call to the NAMESTRING function returns the pathname as a string.
- The call to the SETF macro changes the value of the default directory to DBA1:[SMITH].
- The last two calls to the NAMESTRING function show that the return values of the DEFAULT-DIRECTORY function and the *DEFAULT-PATHNAME-DEFAULTS* variable are still the same.

DEFINE-ALIEN-FIELD-TYPE Macro

Defines alien-structure field types.

For information about alien structures, see Chapter 5 in the *VAX LISP/VMS System Access Guide*.

Format

DEFINE-ALIEN-FIELD-TYPE *name lisp-type primitive-type access-function setf-function*

Arguments

name
The name of the alien-field type being defined.

lisp-type
A LISP data type indicating the type of LISP object to which the field is to be mapped.

primitive-type
Either one of the predefined alien-field types or a type that was previously defined with the DEFINE-ALIEN-FIELD-TYPE macro. A LISP object of type *primitive-type* is extracted from the alien structure's data when the field is accessed. The object is then passed to the specified access function. Predefined alien-field types are listed in Table 5–1 in the *VAX LISP/VMS System Access Guide.*

access-function
A function of one argument (whose type is *primitive-type*) that returns an object of type *lisp-type*.

setf-function

A function of one argument (whose type is *lisp-type*) that returns an object whose type is the type of the default SETF form, as defined by the *primitive-type* argument. When the object is returned, it is packed into the alien structure's field data.

Return Value

The name of the alien-field type.

NOTE

Functions that access and set field values can take more than one argument; additional arguments are optional. When the type argument in the DEFINE-ALIEN-STRUCTURE macro's field description is a list, the first element of the list is the field type, and the remaining elements are expressions that the LISP system evaluates when it evaluates the access function. The resulting values are passed as additional arguments to the functions that access or set the field.

Examples

1.
```
Lisp> (define-alien-field-type integer-string-8
        'integer
        :string
        #'(lambda (x) (parse-integer x :junk-allowed t))
        #'(lambda (x) (format nil "~s" x)))
INTEGER-STRING-8
Lisp> (define-alien-structure two-ascii-integers
        (int-1 integer-string-8 0 8)
        (int-2 integer-string-8 8 16))
TWO-ASCII-INTEGERS
```

   - The call to the DEFINE-ALIEN-FIELD-TYPE macro defines a field type named INTEGER-STRING-8. The field type INTEGER-STRING-8 causes an alien structure to convert strings to integers.
   - The call to the DEFINE-ALIEN-STRUCTURE macro defines an alien structure named TWO-ASCII-INTEGERS that has two fields, each of type INTEGER-STRING-8.

2.
```
Lisp> (define-alien-field-type selection
        t
        :unsigned-integer
        #'(lambda (n) (nth n '(ma ri ny)))
        #'(lambda (x) (position x '(ma ri ny))))
SELECTION
```

This is an example of how the :SELECTION type could be implemented. The example defines an alien-field type named SELECTION. This type defines a relationship between unsigned integers in an alien field and LISP data objects. In accessing the value of a field of this type, the access-function uses the integer stored in the alien field as an index into a list. In setting the value in this type of field, the position of a LISP object in that list is used to define the integer value stored in the alien structure.

DEFINE-ALIEN-STRUCTURE Macro

Defines alien structures. An alien structure is a collection of bytes containing VAX data types.

The syntax of the `DEFINE-ALIEN-STRUCTURE` macro is similar to the `DEFSTRUCT` macro described in *Common LISP: The Language.*

For an explanation of how to define an alien structure, see Chapter 5 in the *VAX LISP/VMS System Access Guide.*

Format

DEFINE-ALIEN-STRUCTURE *name-and-options*
[doc-string]
*{field-description}**

Arguments

name-and-options

The *name-and-options* argument is the name and the options of a new LISP data type. The name argument must be a symbol. The options define the characteristics of the alien structure. If you do not specify options, you can specify the *name-and-options* argument as a symbol:

name

If you specify options, specify the *name-and-options* argument as a list whose first element is the name:

(*name* {(*keyword value*)}*)

Using the following format, specify options as a list of keyword-value pairs.

(*keyword value*)

Table 1 lists the keyword-value pairs that you can specify.

Table 1: DEFINE-ALIEN-STRUCTURE Options

Keyword-Value Pair	Description
`:CONC-NAME` *name*	Names the access functions. The value can be a symbol or `NIL`. If you specify a symbol, the symbol becomes a prefix in the access function names. If you wish to include a hyphen (−) in the access function names, specify it as part of the prefix. If you specify `NIL`, the access function names are the same as the field names. By default, the prefix is the alien structure name followed by a hyphen.

(continued on next page)

Table 1 (Cont.): DEFINE-ALIEN-STRUCTURE Options

Keyword-Value Pair	Description
:CONSTRUCTOR *name*	Names the constructor function. The value can be a symbol or NIL. If you specify a symbol, the symbol becomes the name of the constructor function. If you specify NIL, the macro does not define a constructor function. If you do not specify this keyword, the constructor function's name is the prefix MAKE- attached to the alien structure name.
:COPIER *name*	Names the copier function. The value can be a symbol or NIL. If you specify a symbol, the symbol becomes the name of the copier function. If you specify NIL, the macro does not create a copier function. If you do not specify this keyword, the copier function's name is the prefix COPY- attached to the alien structure name.
:PREDICATE *name*	Names the predicate function. The value can be a symbol or NIL. If you specify a symbol, the symbol becomes the name of the predicate function. If you specify NIL, the macro does not define a predicate function. If you do not specify this keyword, the macro names the predicate function by attaching the structure name to the characters -P.
:PRINT-FUNCTION *function-name*	Specifies the print function for the alien structure. The value must be a function. If you do not specify this keyword, the LISP system displays the alien structure in the following format: #<Alien Structure *name number*> In the preceding format, *name* is the name of the alien structure and *number* is a unique identification number, which distinguishes alien structures that have the same name.

doc-string

The documentation string to be attached to the symbol that names the alien structure. The documentation string is of type STRUCTURE. See *Common LISP: The Language* for information on the DOCUMENTATION function.

field-description

A field description for the alien structure. Specify a field description in the following format:

(*name type start end options*)

The *name* argument must be a symbol. It names functions that access and set the value of the alien structure field.

The *type* argument determines the method by which the VAX data type stored in a field is converted to a LISP object and vice versa. Valid types are:

:STRING
:VARYING-STRING
:SIGNED-INTEGER
:UNSIGNED-INTEGER
:BIT-VECTOR
:F-FLOATING
:G-FLOATING

:D-FLOATING
:H-FLOATING
:POINTER
:SELECTION
Types defined with the VAX LISP DEFINE-ALIEN-FIELD-TYPE macro

See Chapter 5 in the *VAX LISP/VMS System Access Guide* for more information on field types.

As in Common LISP, the *start* and *end* arguments are zero-based, with *start* being inclusive and *end* being exclusive.

The *start* argument must be a rational number or, in some cases, a fixnum (see Section 5.4.3.1 in the *VAX LISP/VMS System Access Guide*) that specifies the 8-bit byte start position of the field in the alien structure's data area. Default: none.

The *end* argument must be a rational number or, in some cases, a fixnum (see Section 5.4.3.1 in the *VAX LISP/VMS System Access Guide*) that specifies the 8-bit byte end position of the field in the alien structure's data area. The last position a field occupies is the position that precedes the field's end position value. Default: none.

The *options* define the characteristics for the field. Specify each option with a keyword-value pair:

keyword value

Table 2 lists the keyword-value pairs that you can specify.

Table 2: DEFINE-ALIEN-STRUCTURE Field Options

Keyword-Value Pair	Description
:DEFAULT *form*	Specifies the default initial value that is to occupy the field. If the field's initial value was not specified in a call to the alien structure's constructor function, the form is evaluated when the constructor function is called. The value that results from the evaluation is the field's default initial value. This value defaults to NIL.
:READ-ONLY *value*	Specifies whether the field can be accessed or set. The value can be T or NIL. If you specify T, the macro generates access functions that are unacceptable place indicators in a call to the SETF macro. If you specify NIL, the macro generates access functions that are acceptable place indicators in a call to the SETF macro. The default is NIL.
:OCCURS *integer*	Specifies the number of times the field is to be represented within the alien structure. The value must be an integer. The default value is 1 (which means no repeats).

(continued on next page)

Table 2 (Cont.): DEFINE-ALIEN-STRUCTURE Field Options

Keyword-Value Pair	Description
:OFFSET *rational-number*	Specifies the distance in 8-bit bytes from the start of one occurrence of the field to the start of the next occurrence of the field. The value must be a rational number. If you specify a value that is greater than the field's length, the alien structure contains gaps. You can access the gaps with other field definitions.

Return Value

The name of the alien structure.

Example

```
Lisp> (define-alien-structure et
        (space-ship    :string 0 10)
        (phone-number  :unsigned-integer 10 17)
        (home          :string 17 32))
ET
```

Defines an alien structure named ET, which contains three fields named SPACE-SHIP, PHONE-NUMBER, and HOME. The fields SPACE-SHIP and HOME are defined to be strings of length 10 and 15, respectively. The field PHONE-NUMBER is defined to be a 7-byte unsigned integer.

For more examples of how to define alien structures, see Chapter 5 in the *VAX LISP/VMS System Access Guide*.

DEFINE-EXTERNAL-ROUTINE Macro

Defines an external routine that a LISP program is to call. You can call routines defined with this macro with the VAX LISP CALL-OUT macro. For information about how to use the VAX LISP callout facility, see Chapter 4 in the *VAX LISP/VMS System Access Guide*.

Format

DEFINE-EXTERNAL-ROUTINE *name-and-options*
[*doc-string*]
{*argument-description*}*

Arguments

name-and-options

The name argument is the name of the external routine that is being defined. It must be a symbol; it may not be the name of a LISP function. The options define the characteristics of the routine. If you do not specify options, you can specify the *name-and-options* argument as a symbol:

name

If you specify options, specify the *name-and-options* argument as a list whose first element is the name:

(*name* {*keyword value*}*)

Specify the options with keyword-value pairs:

keyword value

The option values are not evaluated.

Table 3 lists the keyword-value pairs that you can specify.

Table 3: DEFINE-EXTERNAL-ROUTINE Options

Keyword-Value Pair	Description
`:CHECK-STATUS-RETURN` *value*	Specifies whether the callout facility is to check the severity of the value that an external routine returns in register R0. The value you specify can be `T`, an integer, or `NIL`. If you specify `T`, the callout facility checks the severity of the return value. If the severity is warning, error, or severe, the LISP system signals a continuable error. If you specify an integer, an error is signaled if that value is returned by the routine. If you specify `NIL`, the callout facility does not check the severity of the return value. `NIL` is the default value. If you specify this option, do not specify the `:RESULT` option.
`:ENTRY-POINT` *string*	Names the external routine's entry point. The value must be a string. The macro converts the name to uppercase characters. The default value is the print name of the external routine name.
`:FILE` *pathname*	Specifies the shareable image that was created for the external routine. This must be in uppercase characters and must be a logical name or the name of an executable image in the SYS$SHARE directory. The file specification is merged with the file SYS$SHARE:.EXE. You must specify this option unless you are calling a system service.

(continued on next page)

Table 3 (Cont.): DEFINE-EXTERNAL-ROUTINE Options

Keyword-Value Pair	Description
:RESULT *type*	Specifies the type of LISP object that the external routine is to return. The value can be a LISP type, a type-spec-list, or NIL. A type-spec-list has the following format: :RESULT (:LISP-TYPE *lisp-type* :VAX-TYPE *vax-type*) See Table 4–2 in the *VAX LISP/VMS System Access Guide* for a list of LISP/VAX types. NIL specifies that the routine returns no value. The default value is NIL. If you specify this option, do not specify the :CHECK-STATUS-RETURN option.
:TYPE-CHECK *value*	Specifies whether the callout facility is to check the types of the arguments passed to the external routine for compatibility with the LISP types specified in the argument specification. The value can be T or NIL. If you specify T, the facility checks the types for compatibility; if you specify NIL, the facility does not check the argument types. The default value is NIL.

doc-string

The documentation string for the symbol that names the external routine. The documentation string is of type EXTERNAL-ROUTINE. See *Common LISP: The Language* for information on the DOCUMENTATION function.

argument-description

An argument description that is to be passed to the external routine. Include as many descriptions as the arguments you want to call. Specify the descriptions in the following format:

(*name options*)

The *name* argument must be a unique symbol in the definition or NIL. The *name* identifies the argument and is used in some error messages. If you do not specify options, you can specify the argument-description argument as a symbol:

name

If you specify options, specify the argument as a list whose first element is the name:

(*name* {*keyword value*}*)

The *options* arguments define the characteristics of an argument. Specify the options with keyword-value pairs:

keyword value

The option values are not evaluated.

Table 4 lists the keyword-value pairs you can specify.

Table 4: DEFINE-EXTERNAL-ROUTINE Argument Options

Keyword-Value Pair	Description
`:ACCESS` *value*	Specifies the type of access the external routine needs for the argument. The value can be either `:IN` or `:IN-OUT`. The default value is `:IN`. If you specify `:IN`, the argument can be read but not modified by the external routine. If you specify `:IN-OUT`, the argument can be both read and destructively modified by the external routine.
`:LISP-TYPE` *type*	Specifies the LISP type of the argument value that the callout facility is to pass to the external routine. See Table 4–2 in the *VAX LISP/VMS System Access Guide* for the values you can specify.
`:MECHANISM` *value*	Specifies the argument-passing mechanism the external routine is to expect for the argument. The values you can specify are `:VALUE`, `:REFERENCE`, and `:DESCRIPTOR`. The default value is `:DESCRIPTOR` for `:VAX-TYPE :TEXT` and `:REFERENCE` for other LISP data types.
`:VAX-TYPE` *type*	Specifies the VAX data type of the argument value that the external routine is to return. See Table 4–2 in the *VAX LISP/VMS System Access Guide* for the values you can specify.

Return Value

The symbol that names the external routine.

Example

```
Lisp> (define-external-routine (mth$acosd
                                 :file "MTHRTL"
                                 :result (:lisp-type
                                             single-float
                                          :vax-type
                                             :f-floating))
        "This routine returns the arc cosine
         of an angle in degrees."
        (x :lisp-type single-float
           :vax-type :f-floating))
```

Defines an RTL routine, called MTH$ACOSD, which returns the arc cosine of an angle in degrees. The routine takes one read-only argument, which is an F_floating number, and returns the result as an F_floating number.

For more examples of how to define external routines, see Chapter 4 in the *VAX LISP/VMS System Access Guide*. These examples also show you how to call out to defined external routines.

DEFINE-FORMAT-DIRECTIVE Macro

Defines a directive for use in a FORMAT control string, supplementing the directives supplied with VAX LISP. In a call to FORMAT, specify a directive you have defined in the form:

~/name/

You can also specify colon and at sign modifiers:

~@:/name/

You can also specify one or more parameters:

~n,n/name/

DEFINE-FORMAT-DIRECTIVE provides means for the body of the format directive you define to receive the value of parameters and the presence or absence of colon and at sign modifiers. See *VAX LISP Implementation and Extensions to Common LISP* for more information about defining format directives.

Format

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

Arguments

name
The name of the FORMAT directive defined with this macro.

NOTE

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

arg
A symbol that is bound to the argument to be formatted by the directive.

stream
A symbol that is bound to the stream to which the printing is to be done.

colon-p
A symbol that is bound to T or NIL, indicating whether a colon was specified in the directive.

atsign-p
A symbol that is bound to T or NIL, indicating whether an at sign was specified in the directive.

parameters

One optional argument is allowed for each prefix parameter in the directive. A symbol supplied as a *parameter* argument will be bound to the corresponding prefix parameter if it was specified in the directive. Otherwise, the default value will be used, as with all optional arguments.

forms

Forms which are evaluated to print *argument* to *stream*. The body can begin with a declaration and/or documentation string.

Return Value

The name of the FORMAT directive that has been defined.

Example

```
Lisp> (define-format-directive evaluation-error
                               (symbol stream colon-p atsign-p
                                &optional (severity 0))
        (declare (ignore atsign-p))
        (fresh-line stream)
        (princ (case severity
                 (0     "Warning: ")
                 (1     "Error: ")
                 (2     "Severe Error: "))
               stream)
        (format stream "~:!The symbol ~S ~:_does not have an ~
                        integer value.~%Its value is: ~:_~S~."
                        symbol (symbol-value symbol))
        (when colon-p
          (write-char #\bell stream)))
EVALUATION-ERROR
Lisp> (setf process nil)
NIL
Lisp> (format t "~1:/evaluation-error/" 'process)
Error: The symbol PROCESS does not have an integer value.
       Its value is: NIL
BEEP
```

- This example shows the definition of a FORMAT directive, a use of the directive, and the printed output.
- The prefix parameter 1 in "~1:/EVALUATION-ERROR/" indicates the severity of the error being signaled. The colon produces a beep on the terminal.

DEFINE-GENERALIZED-PRINT-FUNCTION Macro

Defines a function that specifies how any object is to be pretty-printed, regardless of its form. Generalized print functions are effective only when they are enabled (globally or locally) and when pretty-printing is enabled. You can enable a generalized print function globally by using the GENERALIZED-PRINT-FUNCTION-ENABLED-P function, or you can enable it locally by using the WITH-GENERALIZED-PRINT-FUNCTION macro. An enabled generalized print function is used if its predicate evaluates to a non-NIL value.

See *VAX LISP Implementation and Extensions to Common LISP* for more information about generalized print functions.

Format

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

Arguments

name
The name of the generalized print function being defined.

object
A symbol that is bound to the object to be printed.

stream
A symbol that is bound to the stream to which output is to be sent.

predicate
A form. When the generalized print function has been enabled (globally or locally), the system evaluates this form for every object to be pretty-printed. If the form evaluates to non-NIL on the object to be pretty-printed, the generalized print function will be used.

forms
Forms that print *object* to *stream* or take any other action. These forms can refer to the object and stream by means of the symbols used for *object* and *stream*. The body can begin with a declaration and/or documentation string.

Return Value

The name of the generalized print function that has been defined.

Example

```
Lisp> (define-generalized-print-function print-nil-as-list
         (object stream)
         (null object)
         (princ "( )" stream))
PRINT-NIL-AS-LIST
Lisp> (print nil)
NIL
NIL
Lisp> (pprint nil)
NIL
Lisp> (with-generalized-print-function 'print-nil-as-list
         (print nil)
         (pprint nil))
NIL
( )
Lisp> (setf (generalized-print-function-enabled-p
               'print-nil-as-list)
             t)
T
Lisp> (pprint nil)
( )
```

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

DEFINE-LIST-PRINT-FUNCTION Macro

Defines and enables a function to print lists that begin with a specified element. Defined functions are effective only when pretty-printing is enabled. The system checks the first element of each list to be printed for a match. If the first element of a list matches the name of a list-print function, the list is printed according to the format you have defined.

See *VAX LISP Implementation and Extensions to Common LISP* for more information about pretty-printing.

Format

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

Arguments

symbol
The first element of any list to be printed in the defined format.

list
A symbol that is bound to the list to be printed.

stream
A symbol that is bound to the stream on which printing is to be done.

forms
Forms to be evaluated. The forms refer to the list to be printed and the stream by means of the symbols you supply for *list* and *stream*. The body can include declarations. Calls to FORMAT may also be included.

Return Value

The name of the list-print function that has been defined.

Example

```
Lisp> (define-list-print-function my-setq (list stream)
        (format stream
                "~1!~W~^ ~:I~@{~W~^ ~:_~W~^~%~}~."
                list))
MY-SETQ
Lisp> (setf base '(my-setq hi 3 bye 4))
(MY-SETQ HI 3 BYE 4)
Lisp> (print base)
(MY-SETQ HI 3 BYE 4)
(MY-SETQ HI 3 BYE 4)
Lisp> (pprint base)
(MY-SETQ HI 3
         BYE 4)
```

- The list-print function MY-SETQ is defined.
- The call to PRINT does not use the list-print function MY-SETQ to print the value of BASE, because pretty-printing is not enabled.
- The call to PPRINT does use the list-print function MY-SETQ to print the value of BASE.

DELETE-PACKAGE Function

Uninterns all symbols interned in the package, unuses all packages the package uses, and deletes the package. An error is signaled if any other package uses the package.

Format

DELETE-PACKAGE *package*

Argument

package
A package, or a string or symbol naming a package.

Return Value

`T.`

Example

```
Lisp> (delete-package "test-package")
T
Lisp> (find-package "test-package")
NIL
```

DESCRIBE Function

Displays information about a specified object. If the specified object has a documentation string, this function displays the string in addition to the other information the function displays. The type of information the function displays depends on the type of the object. For example, if a symbol is specified, the function displays the symbol's value, definition, properties, and other types of information. If a floating-point number is specified, the number's internal representation is displayed in a way that is useful for tracking such things as roundoff errors.

Format

DESCRIBE *object*

Argument

object
The object about which information is to be displayed.

Return Value

No value.

Examples

1. `Lisp> (describe 'c)`

```
It is the symbol C
Package:  USER
Value:    unbound
Function: undefined
```

2. `Lisp> (describe 'factorial)`

```
It is the symbol FACTORIAL
Package:  USER
Value:    unbound
Function: a compiled-function
    FACTORIAL n
```

3. `Lisp> (describe pi)`

```
It is the long-float 3.14159265358979323846264338327950L0
Sign:         +
Exponent:     2 (radix 2)
Significand:  0.785398163397448309615660845819880L0
```

4. `Lisp> (describe '#(1 2 3 4 5))`

```
It is a simple-vector
Dimensions:    (5)
Element type: t
Adjustable:    no
Fill Pointer:  no
Displaced:     no
```

Displays information about the simple-vector `#(1 2 3 4 5)`.

DIRECTORY Function

Converts its argument to a pathname and returns a list of the pathnames for the files matching the specification. The DIRECTORY function is similar to the DCL DIRECTORY command.

Format

DIRECTORY *pathname*

Argument

pathname
The pathname, namestring, stream, or symbol for which the list of file system pathnames is to be returned. In VAX LISP/VMS, this argument is merged with the following default file specification:

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

The *host*, *device*, and *directory* values are supplied by the *DEFAULT-PATHNAME-DEFAULTS* variable.

Specifying just a directory is equivalent to specifying a directory with wildcards (*) in the name, type, and version fields of the argument. For example, the following two expressions are equivalent:

```
(directory "[mydirectory]")
(directory "[mydirectory]*.*;*")
```

Both expressions return a list of pathnames that represent the files in the directory MYDIRECTORY.

Specifying a directory with just a specified version field is equivalent to specifying a directory and version with wildcards (*) in the name and type fields of the argument. For example, the following two expressions are equivalent:

```
(directory "[mydirectory];0")
(directory "[mydirectory]*.*;")
```

Both expressions return a list of the pathnames that represent the newest versions of the files in the directory MYDIRECTORY.

The following equivalent expressions return the list of pathnames for files in your default directory:

```
(directory "")
(directory (default-directory))
```

Return Value

A list of pathnames, if the specified pathname is matched, or NIL, if the pathname is not matched.

Example

```
Lisp> (defun my-directory (&optional (filename ""))
        (let ((pathname (pathname filename))
              (directory (directory filename)))
          (cond ((null directory)
                 (format t
                         "~%No files match ~A.~%"
                         (namestring filename)))
                (t (format t
                        "~%The following ~:[files are ~;file is ~]~
                        in the directory ~A:[~A]:"
                           (equal (length directory) 1)
                           (pathname-device
                            (nth 0 directory))
                           (pathname-directory
                            (nth 0 directory)))
                   (dolist (x directory)
                     (format t "~&~2T~A" (file-namestring x)))
                   (terpri)))
          (values)))
MY-DIRECTORY
Lisp> (my-directory)
The following files are in the directory DBA1:[SMITH.TESTS]:
  TEST5.DRB;1
  TEST1.LSP;7
  TEST1.LSP;6
  TEST1.LSP;5
  EXAMPLE.TXT;2
  TEST3.LSP;15
  TEST6.LSP;1
Lisp> (my-directory ".lsp;")
The following files are in the directory DBA1:[SMITH.TESTS]:
  TEST1.LSP;7
  TEST3.LSP;15
  TEST6.LSP;1
```

- The call to the DEFUN macro defines a function that formats the output of the DIRECTORY function, making the output more readable. The function is defined such that it accepts an optional argument and does not return a value.
- The first call to the function MY-DIRECTORY shows how the function formats the directory output when an argument is not specified.
- The second call to the function MY-DIRECTORY includes an argument; the output includes only the latest versions of file names of file type .LSP.

DRIBBLE Function

Echoes the input and output of an interactive LISP session to a specified file, enabling you to save a record of what you do during the session in the form of a file.

When you want to stop the DRIBBLE function from echoing input and output to the pathname, close the file by calling the DRIBBLE function without an argument.

In VAX LISP/VMS, there are two restrictions on the use of the DRIBBLE function:

- When you are in the VAX LISP Editor, terminal I/O is not recorded in a dribble file.
- In the DECwindows-based development environment, I/O to windows other than the Listener is not recorded in a dribble file.

Format

DRIBBLE &OPTIONAL *pathname*

Argument

pathname
The pathname to which the input and output of the LISP session is to be sent.

Return Value

If an argument is specified with the function, no value is returned and dribbling is turned on. If dribbling is on and the function is called with no arguments, T is returned and dribbling is turned off. If dribbling is off and the function is called without an argument, NIL is returned.

Examples

1.
```
Lisp> (dribble "newfunction.txt")
Dribbling to DBA1:[SMITH]NEWFUNCTION.TXT;1
Lisp>
```

Creates a dribble file named NEWFUNCTION.TXT. The LISP system sends input and output to the file until you call the DRIBBLE function again (without an argument) or exit LISP.

2.
```
Lisp> (dribble)
T
```

Closes the dribble file that was previously opened and turns dribbling off.

DYNAMIC-SPACE-RATIO Function

Returns the percentage of dynamic space that may be filled before a full garbage collection is performed.

The ratio may be changed by using the SETF macro. The new value must be a floating-point number greater than 0 and less than or equal to 1. Setting this ratio may help the memory management system make more efficient use of memory. However, the ratio is only a guideline: it may be reset to more appropriate values when the memory management system finds the current value to be inappropriate for existing conditions.

The default dynamic space ratio is 0.5. Setting the dynamic space ratio to a higher value decreases the frequency of garbage collections. Values less than 0.5 are probably wasteful of space.

Format

DYNAMIC-SPACE-RATIO

Argument

None.

Return Value

A floating-point number.

Example

```
Lisp> (dynamic-space-ratio)
0.5
Lisp> (setf (dynamic-space-ratio) .75)
0.75
```

ED Function

Invokes the VAX LISP Editor. This function can be specified with an optional argument whose value can be a namestring, pathname, or symbol. In VAX LISP, the argument's value can also be a list. In addition, you can specify a :TYPE argument whose value can be the :FUNCTION or :VALUE keyword.

NOTE

If you bind a control character, such as Ctrl/E, to the ED function using BIND-KEYBOARD-FUNCTION, specify an interrupt level of 1, the default, or 0 with the :LEVEL keyword. Do not specify a higher interrupt level.

ED Function

See Chapter 3 of the *VAX LISP/VMS Program Development Guide* for information on using the VAX LISP Editor.

Format

ED &OPTIONAL *x* &KEY :TYPE

Arguments

x
The namestring, pathname, symbol, or list that is to be edited. If you specify a list, the list must be a generalized variable that can be specified in a call to the SETF macro. The list is evaluated, and a value that you can edit is returned. When you write the buffer containing the value, the Editor replaces the value of the generalized variable with the new value.

If you specify a symbol, you can also specify the keyword argument. The value of the keyword informs the Editor whether you want to edit the symbol's function or macro definition or its value.

:TYPE
You can specify this argument if the *x* argument is a symbol. The value is a keyword that affects the interpretation of the *x* argument's value. You can specify one of the following keywords:

`:FUNCTION`	The Editor is invoked to edit the function or macro definition associated with the specified symbol.
`:VALUE`	The Editor is invoked to edit the specified symbol's value.

The default value for the `:TYPE` keyword is the `:FUNCTION` keyword.

Return Value

No value.

Examples

1.
```
Lisp> (ed "[smith.lisp]newprog.lsp")
```

   Invokes the Editor to edit the file NEWPROG.LSP in the directory [SMITH.LISP].

2.
```
Lisp> (ed 'factorial)
```

   Invokes the Editor to edit a function named `FACTORIAL`.

3.
```
Lisp> (ed 'gameboard-array :type :value)
```

   Invokes the Editor to edit the value of the symbol `GAMEBOARD-ARRAY`.

4.

```
Lisp> (defstruct room
        doors
        windows
        outlets
        color)
ROOM
Lisp> (setq room2 (make-room :doors 1
                             :windows 3
                             :outlets 4
                             :color 'blue))
#S(ROOM :DOORS 1 :WINDOWS 3 :OUTLETS 4 :COLOR BLUE)
Lisp> (ed '(room-color room2))
```

- The call to the DEFSTRUCT macro defines a structure named ROOM.
- The call to the SETQ special form creates an instance of the structure ROOM.
- The call to the ED function invokes the Editor to edit the COLOR slot of the structure bound to ROOM2.

ENLARGE-BINDING-STACK Function

Enlarges the VAX LISP binding stack by the specified number of pages. Use this function if the default size of the binding stack is too small to accommodate a large or complex program.

Format

ENLARGE-BINDING-STACK *number-of-pages*

Argument

number-of-pages
The number of 512-byte pages by which to enlarge the binding stack.

Return Value

Unspecified.

ENLARGE-LISP-MEMORY Function

Increases the amount of virtual memory allocated to VAX LISP by asking the operating system for a specified number of 64K-byte segments.

Format

ENLARGE-LISP-MEMORY *segments*

ENLARGE-LISP-MEMORY Function

Argument

segments
The number of 64K-byte segments by which to enlarge VAX LISP memory.

Return Value

The specified number of segments.

Example

```
Lisp> (area-segment-limit :dynamic)
100
Lisp> (enlarge-lisp-memory 50)
50
Lisp> (area-segment-limit :dynamic)
150
```

ERROR-ACTION Variable

Determines the action that the VAX LISP Error Handler is to take when an unhandled error occurs. The value of this variable can be the `:EXIT` or the `:DEBUG` keyword. If the value is `:EXIT`, the Error Handler causes the LISP system to exit; if the value is `:DEBUG`, the Error Handler invokes the VAX LISP Debugger. The default value is `:DEBUG` for interactive LISP sessions; otherwise, the default value is `:EXIT`.

Besides setting this variable within a LISP form, you can also set it on LISP initialization with the /ERROR_ACTION qualifier. See Chapter 2 of the *VAX LISP/VMS Program Development Guide.*

Example

```
Lisp> (car 'a)

Error in CAR: Argument must be a list: A.

Control Stack Debugger
Apply #6: (CAR A)
Debug 1> quit
Lisp> (setf *error-action* :exit)
:EXIT
Lisp> (car 'a)

Error in CAR: Argument must be a list: A.
$
```

- When the first error occurs, the LISP system invokes the VAX LISP Debugger because the value of the `*ERROR-ACTION*` variable is `:DEBUG` (the default).
- The call to the `SETF` macro sets the value of the variable to `:EXIT`.
- The second time the error occurs, the LISP system exits and control returns to the VMS command level.

EXIT Function

Invokes the VMS Exit system service, after unwinding the stack, causing the LISP system to exit and to return control to the VMS command level.

You can pass the status of the LISP system to the VMS command level when you exit the LISP system by specifying an optional argument. When the LISP system exits, the argument's value is passed to the VMS command level.

Format

EXIT &OPTIONAL *status*

Argument

status
A fixnum or a keyword that indicates the status of the LISP system that is to be returned to the VMS command level when the LISP system exits. The keywords you can specify and the types of status they return are:

`:ERROR`	Error status
`:SUCCESS`	Success status
`:WARNING`	Warning status

Return Value

The `EXIT` function does not return to LISP.

Examples

1.
```
Lisp> (exit)
$
```

Exits the LISP system.

2.
```
Lisp> (exit :error)
$ show symbol $status
  $STATUS = "%X112D8012"
```

Exits the LISP system. When control returns to the VMS command level, the VAX LISP exit status contains an error status.

FORCE-INTERRUPT-FUNCTION Function

Forces an AST and thus the invocation of the related interrupt function specified by its argument. FORCE-INTERRUPT-FUNCTION is primarily useful for debugging.

Format

FORCE-INTERRUPT-FUNCTION *iif-id*

Argument

iif-id
An interrupt function identifier previously returned by an INSTATE-INTERRUPT-FUNCTION function.

Return Value

Unspecified.

Example

```
Lisp> (defun timer-interrupt-handler ()
         (princ "The timer has expired"))
TIMER-INTERRUPT-HANDLER
Lisp> (setf timer-iif (instate-interrupt-function
                           #'timer-interrupt-handler))
8454198
Lisp> (force-interrupt-function timer-iif)
The timer has expired
T
```

- The function TIMER-INTERRUPT-HANDLER is instated as an interrupt function whose *iif-id* is retained as the value of TIMER-IIF.
- Passing TIMER-IIF in a call to FORCE-INTERRUPT-FUNCTION causes TIMER-INTERRUPT-HANDLER to execute.

Format Directives Provided with VAX LISP

VAX LISP provides eight directives for the FORMAT function, in addition to those described in *Common LISP: The Language*. Table 5 lists and describes these directives. See *VAX LISP Implementation and Extensions to Common LISP* for more information about using these directives.

Table 5: Format Directives Provided with VAX LISP

<table>
<tr><th>Directive</th><th colspan="2">Effect</th></tr>
<tr><td>~W</td><td colspan="2">Prints the corresponding argument under direction of the current print variable values. The argument for ~W can be any LISP object. This directive takes a colon modifier and four prefix parameters.
Use the colon modifier (~:W) when you want to set *PRINT-PRETTY* and *PRINT-ESCAPE* to T, but *PRINT-LENGTH*, *PRINT-LEVEL*, and *PRINT-LINES* to NIL.
The four prefix parameters specify padding.
~*mincol,colinc,minpad,padchar*W
These parameters are identical to those used with the Common LISP ~A directive:</td></tr>
<tr><td></td><td><i>mincol</i></td><td>Specifies the minimum width of the printed representation of the object. FORMAT inserts padding characters on the right, until the width is at least <i>mincol</i> columns. Use the at sign modifier with <i>minpad</i> to insert the padding characters on the left instead. The default for <i>mincol</i> is 0.</td></tr>
<tr><td></td><td><i>colinc</i></td><td>Specifies an increment: the number of padding characters to be inserted at one time until the width is at least <i>mincol</i> columns. The default is 1.</td></tr>
<tr><td></td><td><i>minpad</i></td><td>Specifies the minimum number of padding characters to be inserted. The default is 0.</td></tr>
<tr><td></td><td><i>padchar</i></td><td>Preceded by a single quote, specifies the padding character. The default is the space character.</td></tr>
<tr><td>~!</td><td colspan="2">Begins a logical block. A logical block is a hierarchical grouping of FORMAT directives treated as a unit. FORMAT must be processing a logical block with *PRINT-PRETTY* true to enable pretty-printing. Directives inside a logical block refer to elements of a single list taken from the argument list to FORMAT. (If the argument supplied to the logical block is not a list, the logical block is skipped and the argument is printed as if with ~W.) The logical block directive takes colon and at sign modifiers.
When the logical block directive is modified by a colon (~:!), the directive sets *PRINT-PRETTY* and *PRINT-ESCAPE* to T but *PRINT-LENGTH*, *PRINT-LEVEL*, and *PRINT-LINES* to NIL.
When the logical block directive is modified by an at sign (~@!), directives within the logical block take successive arguments from the FORMAT argument list. The logical block uses up all the arguments, not just a single list argument. Arguments not needed by the logical block are used up as well, so that they are not available for subsequent directives.
Specify a parameter of 1 (~1!) to enclose the output in parentheses.</td></tr>
<tr><td>~.</td><td colspan="2">Ends a logical block. If modified by an at sign (~@!), the directive inserts a new line if needed after every blank space character.</td></tr>
</table>

(continued on next page)

Table 5 (Cont.): Format Directives Provided with VAX LISP

Directive	Effect
~_	Specifies a multiline mode new line and marks a logical block section. This directive takes colon and at sign modifiers. When modified by a colon (~:_), the directive starts a new line if not enough space is on the line to print the next logical block section. When modified by an at sign (~@_), the directive starts a new line if miser mode is enabled. The ~_ directive and its variants are effective only when used within a logical block with pretty-printing enabled.
~*n*I	Sets indentation for subsequent lines to *n* columns after the beginning of the logical block or after the prefix. When modified by a colon (~*n*:I), the directive causes FORMAT to indent subsequent lines *n* spaces from the column corresponding to the position of the directive. The ~*n*I directive and the ~*n*:I variant are effective only when used within a logical block with pretty-printing enabled.
~*n*/FILL/	Prints the elements of a list with as many elements as possible on each line. If *n* is 1, FORMAT encloses the printed list in parentheses. If pretty-printing is disabled, the directive causes FORMAT to print the output on a single line.
~*n*/LINEAR/	If the elements of the list to be printed cannot be printed on a single line, this directive prints each element on a separate line. If *n* is 1, FORMAT encloses the printed list in parentheses. If pretty-printing is disabled, this directive causes FORMAT to print the output on a single line.
~*n*,*m*/TABULAR/	Prints the list in tabular form. If *n* is 1, FORMAT encloses the list in parentheses; *m* specifies the column spacing. If pretty-printing is disabled, this directive causes FORMAT to print the output on a single line.

GC Function

Invokes a full garbage collection. The LISP system automatically initiates garbage collection during normal system use whenever necessary. You might want to use the GC function to invoke the garbage collector just before a time-critical part of a LISP program. Using the GC function this way reduces the possibility of the LISP system initiating a garbage collection when a critical part of the program is executing.

NOTE

> The LISP system does not use the GC function to initiate garbage collections. Therefore, redefining the GC function does not prevent garbage collection. You can disable garbage collecting by using the GC-MODE function.

See *VAX LISP Implementation and Extensions to Common LISP* for a description of the garbage collector.

GC Function

Format

GC

Argument

None.

Return Value

T, when garbage collection is completed.

Example

```
Lisp> (gc)
; Starting full GC ...
; ... Full GC finished
T
```

Invokes the garbage collector. Whether the messages are printed when a garbage collection occurs depends on the value of the *GC-VERBOSE* variable.

GC-COUNT Function

Returns the number of garbage collections performed since the LISP image was invoked. You may specify the type of garbage collection to be counted.

Format

GC-COUNT &OPTIONAL *type*

Argument

type
Possible values are:

0, 1, or 2	Collections in the three ephemeral areas, respectively.
:FULL	Full collections.
:DEFAULT	Both full and ephemeral collections. This is the default.

A *type* of T is the same as :DEFAULT.

Return Value

An integer.

Example

```
Lisp> (gc-count 0)
12
Lisp> (gc-count :full)
1
```

The number of ephemeral collections is higher than the number of full garbage collections in this session.

GC-MODE Function

Returns or sets the mode of garbage collection, depending on the value of the *type* argument. You can use this function to control the collection algorithm used or to prevent garbage collections. See the *VAX LISP Implementation and Extensions to Common LISP* manual for details on VAX LISP garbage collection.

Format

GC-MODE &OPTIONAL *type*

Argument

type
Determines whether the function returns or sets the garbage collection mode. Possible values are:

`:DEFAULT`	The function returns the current mode. This is the default.
`:NONE`	Garbage collection is disabled.
`:FULL`	The full garbage collector is enabled.
`:EPHEMERAL`	The VAX LISP system attempts to enable the ephemeral garbage collector. The attempt may fail due to insufficient memory. `:EPHEMERAL` always implies `:FULL`.

Values of `T` or `NIL` for *type* are also valid and cause the function to return the current mode.

Return Value

One of three keywords indicating which type of garbage collection may occur: `:NONE`, if garbage collecting is disabled; `:FULL`, if full garbage collection only is enabled; or `:EPHEMERAL`, if both ephemeral and full garbage collections are enabled.

Examples

1.
```
Lisp> (gc-mode)
:FULL
```

The ephemeral collector has been shut off; only the full collector is enabled

2.
```
Lisp> (enlarge-lisp-memory 40)
40
Lisp> (gc-mode :ephemeral)
:EPHEMERAL
```

With more memory allocated by the ENLARGE-LISP-MEMORY function, both ephemeral and full stop-and-copy garbage collections are enabled by this call to the GC-MODE function.

GC-VERBOSE Variable

A variable whose value is used as a flag to determine whether the LISP system is to display messages when a full garbage collection occurs. If the flag is NIL, the system displays no messages. If the flag is not NIL, the system displays a message before and after a garbage collection occurs. The default value is T. The VAX LISP *PRE-GC-MESSAGE* and *POST-GC-MESSAGE* variables control the contents of the messages displayed.

The ephemeral garbage collector does not display messages.

For more information on garbage collector messages, see the *VAX LISP Implementation and Extensions to Common LISP* manual.

Examples

1.
```
Lisp> *gc-verbose*
T
Lisp> (gc)
; Starting full GC ...
; ... Full GC finished
T
```

In this example, *GC-VERBOSE* has the default value T, so the LISP system displays a message before and after a garbage collection occurs.

The text of the messages depends on the values of the *PRE-GC-MESSAGE* and *POST-GC-MESSAGE* variables. These are the default messages.

2.
```
Lisp> (setf *gc-verbose* nil)
NIL
Lisp> (gc)
T
```

The second call to the GC function shows that the system does not display messages when the value of *GC-VERBOSE* is NIL.

GENERALIZED-PRINT-FUNCTION-ENABLED-P Function

Used to enable globally a generalized print function or to test whether a generalized print function is enabled. GENERALIZED-PRINT-FUNCTION-ENABLED-P is a predicate, and it can be used as a *place* form with SETF.

See *VAX LISP Implementation and Extensions to Common LISP* for more information about using generalized print functions.

Format

GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*

Argument

name
A symbol identifying the generalized print function to be enabled or tested.

Return Value

T or NIL.

Example

```
Lisp> (generalized-print-function-enabled-p 'print-nil-as-list)
NIL
Lisp> (define-generalized-print-function print-nil-as-list
               (object stream)
               (null object)
         (princ "( )" stream))
PRINT-NIL-AS-LIST
Lisp> (setf (generalized-print-function-enabled-p
               'print-nil-as-list)
       t)
T
Lisp> (pprint nil)
( )
```

- The first use of the GENERALIZED-PRINT-FUNCTION-ENABLED-P function returns NIL, because no generalized print function named PRINT-NIL-AS-LIST has been defined.
- The call to the DEFINE-GENERALIZED-PRINT-FUNCTION macro defines the generalized print function PRINT-NIL-AS-LIST.

- The call to `SETF` globally enables the generalized print function `PRINT-NIL-AS-LIST`.
- The `PPRINT` call prints `( )`, because the generalized print function is enabled globally and pretty-printing is enabled.

GET-DEVICE-INFORMATION Function

Returns information about a device. The keywords you specify with the function determine the type of information the function returns.

This function is similar to the VMS system service $GETDVI. For more information on the $GETDVI system service, see the *VMS System Services Reference Manual* and the *VMS I/O User's Reference Manual: Part I.*

Format

GET-DEVICE-INFORMATION *device* **&REST** *keyword*

Arguments

device
The string that names the device about which information is to be returned.

keyword
Optional keywords that specify types of information about the specified device. Do not specify values with the keywords.

Table 6 lists the keywords that you can specify and the values they return.

Table 6: GET-DEVICE-INFORMATION Keywords

Keyword	Return Value
`:ACP-PID`	An integer that specifies the ACP process ID.
`:ACP-TYPE`	An integer that specifies the ACP type code.
`:BUFFER-SIZE`	An integer that specifies the buffer size.
`:CLUSTER-SIZE`	An integer that specifies the volume cluster size.
`:CYLINDERS`	An integer that specifies the number of cylinders on the device.
`:DEVICE-CHARACTERISTICS`	A vector of 32 bits that specifies the device characteristics. See the *VMS I/O User's Reference Manual: Part I* for information about device characteristics.
`:DEVICE-CLASS`	An integer that specifies the device class.
`:DEVICE-DEPENDENT-0`	A bit vector that specifies device-dependent information.
`:DEVICE-DEPENDENT-1`	A bit vector that specifies device-dependent information.

(continued on next page)

Table 6 (Cont.): GET-DEVICE-INFORMATION Keywords

Keyword	Return Value
:DEVICE-NAME	A string that specifies the device name.
:DEVICE-TYPE	An integer that specifies the device type.
:ERROR-COUNT	An integer that specifies the number of errors that have occurred on the device.
:FREE-BLOCKS	An integer that specifies the number of free blocks on the device; otherwise, NIL.
:LOGICAL-VOLUME-NAME	A string that specifies the logical name associated with the volume on the device. This keyword is valid only for disks.
:MAX-BLOCKS	An integer that specifies the maximum number of logical blocks that can exist on the device.
:MAX-FILES	An integer that specifies the maximum number of files that can exist on the device.
:MOUNT-COUNT	An integer that specifies the number of times the device has been mounted.
:NEXT-DEVICE-NAME	A string that specifies the name of the next volume in the volume set.
:OPERATION-COUNT	An integer that specifies the number of operations that have been performed on the device.
:OWNER-UIC	An integer that specifies the UIC of the owner.
:PID	An integer that specifies the process ID of the owner.
:RECORD-SIZE	An integer that specifies the blocked record size.
:REFERENCE-COUNT	An integer that specifies the number of channels assigned to the device.
:ROOT-DEVICE-NAME	A string that specifies the name of the root volume in the volume set.
:SECTORS	An integer that specifies the number of sectors per track.
:SERIAL-NUMBER	An integer that specifies the serial number.
:TRACKS	An integer that specifies the number of tracks per cylinder.
:TRANSACTION-COUNT	An integer that specifies the number of files open on the device.
:UNIT	An integer that specifies the unit number.
:VOLUME-COUNT	An integer that specifies the number of volumes in the volume set.
:VOLUME-NAME	A string that specifies the name of the volume on the device.
:VOLUME-NUMBER	An integer that specifies the number of the volume on the device.
:VOLUME-PROTECTION	A vector of 32 bits that specifies the volume protection mask.

Return Value

The keywords and their values are returned as a property list in the following format:

(:keyword-1 value-1 :keyword-2 value-2 . . .)

The function preserves the order of the keyword-value pairs in the argument list.

If you do not specify keywords, the function returns a list of all the keyword-value pairs. If the device does not exist, the function returns `NIL`.

Example

```
Lisp> (get-device-information "dba1"
                              :device-name
                              :error-count
                              :mount-count)
(:DEVICE-NAME "_DBA1:" :ERROR-COUNT 0 :MOUNT-COUNT 1)
```

Returns the device name, the error count, and the mount count for the device DBA1.

GET-FILE-INFORMATION Function

Returns information about a file. The keywords that you specify with the function determine the type of information that the function returns. The keywords correspond to VMS RMS file access block (FAB) and extended attribute block (XAB) fields. See the *VMS Record Management Services Manual* for information on FAB and XAB fields.

Format

GET-FILE-INFORMATION *pathname* **&REST** {*keyword*}*

Arguments

pathname
A pathname, namestring, symbol, or stream that represents the name of the file about which information is to be returned.

keyword
Optional keywords that return specific types of information about the specified file. Do not specify values with the keywords.

Table 7 lists the keywords that you can specify and the values they return.

Table 7: GET-FILE-INFORMATION Keywords

Keyword	Return Value
:ALLOCATION-QUANTITY	An integer that specifies the number of blocks allocated for the file.
:BACKUP-DATE	The last universal date and time the file was backed up. If the file has not been backed up, the function returns NIL.
:BLOCK-SIZE	An integer that specifies the block size.
:CREATION-DATE	The universal date and time the file was created.
:DEFAULT-EXTENSION	An integer that specifies the number of blocks added to the file's size when the file was extended.
:END-OF-FILE-BLOCK	An integer that specifies the block in which the file ends.
:EXPIRATION-DATE	The universal date and time the file expires. If an expiration date is not recorded, the function returns NIL.
:FIRST-FREE-BYTE	An integer that specifies the offset of the first byte in the file's end-of-file block.
:FIXED-CONTROL-SIZE	An integer that specifies the fixed control area size.
:GROUP	An integer that specifies the owner group number.
:LONGEST-RECORD-LENGTH	An integer that specifies the length of the longest record in the file.
:MAX-RECORD-SIZE	An integer that specifies the maximum size allowed for a record.
:MEMBER	An integer that specifies the owner member number.
:ORGANIZATION	An integer that specifies the organization.
:PROTECTION	A vector of 16 bits that specifies the protection code.
:RECORD-ATTRIBUTES	An integer that specifies the record attributes.
:RECORD-FORMAT	An integer that specifies the record format.
:REVISION	An integer that specifies the revision number.
:REVISION-DATE	The last universal date and time the file was revised.
:UIC	An integer that specifies the owner UIC.
:VERSION-LIMIT	An integer that specifies the maximum version number the file can have.

Return Value

The keywords and their values are returned as a property list in the following format:

(*:keyword-1 value-1 :keyword-2 value-2 . . .*)

The function preserves the order of the keyword-value pairs in the argument list. If you do not specify keywords, the function returns a list of all the keyword-value pairs. If the file does not exist, the function returns NIL.

Examples

1.
```
Lisp> (get-file-information "important.dat"
                           :allocation-quantity
                           :backup-date)
(:ALLOCATION-QUANTITY 252 :BACKUP-DATE 2654202351)
```

Returns the allocation quantity and backup date for the file IMPORTANT.DAT.

2.
```
Lisp> (defun show-file-size (file)
        (let ((size-list
               (get-file-information file
                                     :allocation-quantity
                                     :end-of-file-block)))
          (format t
                  "~A ~%~
                   ~3T Blocks allocated: ~D~%~
                   ~3T Blocks used:      ~D~%"
                  (namestring (truename file))
                  (getf size-list :allocation-quantity)
                  (getf size-list :end-of-file-block))))
SHOW-FILE-SIZE
Lisp> (show-file-size "myfile.txt")
DBA1:[SMITH]MYFILE.TXT;4
   Blocks allocated: 240
   Blocks used:      239
NIL
```

- The call to the DEFUN macro defines a function named SHOW-FILE-SIZE, which displays the amount of space that is allocated for a specified file and the amount of space the file uses.
- The call to SHOW-FILE-SIZE displays the amount of space that is allocated for the file MYFILE.TXT and the amount of space the file uses.

GET-GC-REAL-TIME Function

Lets you inspect the elapsed time used by the garbage collector during program execution. This function is useful for tuning programs.

The function measures its value in terms of the INTERNAL-TIME-UNITS-PER-SECOND constant. This value is cumulative. It includes the elapsed time used for all the garbage collections that have occurred. For a description of the INTERNAL-TIME-UNITS-PER-SECOND constant, see *Common LISP: The Language*.

When a suspended system is resumed, the elapsed time is set to 0.

For more information on the garbage collector, see *VAX LISP Implementation and Extensions to Common LISP*.

Format

GET-GC-REAL-TIME

GET-GC-REAL-TIME Function

Argument

None.

Return Value

The real time that has been used by the garbage collector.

Examples

1.

```
Lisp> (get-gc-real-time)
3485700000
Lisp> (gc)
; Starting full GC ...
; ... Full GC finished
T
Lisp> (get-gc-real-time)
401210000
```

- The first call to the GET-GC-REAL-TIME function returns the real time used by the garbage collector.
- The call to the GC function invokes a garbage collection.
- The second call to the GET-GC-REAL-TIME function returns the updated real time that has been used by the garbage collector.

2.

```
Lisp> (defmacro gc-elapsed-time (form)
        `(let* ((start-gc (get-gc-real-time))
                (value ,form)
                (end-gc (get-gc-real-time)))
           (format *trace-output*
                "~%GC elapsed time: ~D seconds~%"
                (truncate
                  (- end-gc start-gc)
                  internal-time-units-per-second))))
GC-ELAPSED-TIME
Lisp> (gc-elapsed-time (suspend "myfile.sus"))
; Starting full GC ...
; ... Full GC finished
GC elapsed time: 54 seconds
NIL
```

- The call to the DEFMACRO macro defines a macro named GC-ELAPSED-TIME, which evaluates a form and displays the amount of elapsed time that was used by the garbage collector during a form's evaluation.
- The call to the GC-ELAPSED-TIME function displays the amount of elapsed time the garbage collector used when the LISP system evaluated the form (suspend "myfile.sus").

GET-GC-RUN-TIME Function

Lets you inspect the CPU time used by the garbage collector during program execution. This function is useful for tuning programs.

The function measures its value in terms of the INTERNAL-TIME-UNITS-PER-SECOND constant. This value is cumulative. It includes the CPU time used for all the garbage collections that have occurred. For a description of the INTERNAL-TIME-UNITS-PER-SECOND constant, see *Common LISP: The Language*.

When a suspended system is resumed, the CPU time is set to 0.

For more information on the garbage collector, see *VAX LISP Implementation and Extensions to Common LISP*.

Format

GET-GC-RUN-TIME

Argument

None.

Return Value

The CPU time that has been used by the garbage collector.

Examples

1.
```
Lisp> (get-gc-run-time)
6933
Lisp> (gc)
; Starting full GC ...
; ... Full GC finished
T
Lisp> (get-gc-run-time)
8423
```

- The first call to the GET-GC-RUN-TIME function returns the CPU time used by the garbage collector.
- The call to the GC function invokes a garbage collection.
- The second call to the GET-GC-RUN-TIME function returns the updated CPU time that has been used by the garbage collector.

2.
```
Lisp> (defmacro gc-cpu-time (form)
        `(let* ((start-gc (get-gc-run-time))
                (value ,form)
                (end-gc (get-gc-run-time)))
           (format *trace-output*
                   "~%GC CPU time: ~D seconds~%"
                   (truncate
                     (- end-gc start-gc)
                     internal-time-units-per-second))))
GC-CPU-TIME
Lisp> (gc-cpu-time (suspend "myfile.sus"))
; Starting full GC ...
; ... Full GC finished
GC CPU time: 10 seconds
NIL
```

- The call to the DEFMACRO macro defines a macro named GC-CPU-TIME, which evaluates a form and displays the amount of CPU time that was used by the garbage collector during a form's evaluation.
- The call to the GC-CPU-TIME function displays the amount of CPU time the garbage collector used when the LISP system evaluated the form (suspend "myfile.sus").

GET-INTERNAL-RUN-TIME Function

Returns an integer that represents the elapsed CPU time used for the current process. The function value is measured in terms of the INTERNAL-TIME-UNITS-PER-SECOND constant. For a description of the INTERNAL-TIME-UNITS-PER-SECOND constant, see *Common LISP: The Language*.

Format

GET-INTERNAL-RUN-TIME

Argument

None.

Return Value

The elapsed CPU time used for the current process.

Example

```
Lisp> (defmacro my-time (form)
        `(let* ((start-real-time (get-internal-real-time))
                (start-run-time (get-internal-run-time))
                (value ,form)
                (end-run-time (get-internal-run-time))
                (end-real-time (get-internal-real-time)))
           (format *trace-output*
                  "~&Run Time: ~,2F sec., ~
                   Real Time: ~,2F sec.~%"
                  (/ (- end-run-time start-run-time)
                     internal-time-units-per-second)
                  (/ (- end-real-time start-real-time)
                     internal-time-units-per-second))
          value))
MY-TIME
```

Defines a macro that displays timing information about the evaluation of a specified form.

GET-INTERRUPT-FUNCTION Function

Returns information about the interrupt function specified by its argument.

Format

GET-INTERRUPT-FUNCTION *iif-id*

Argument

iif-id
An interrupt function identifier previously returned by INSTATE-INTERRUPT-FUNCTION.

Return Values

Four values:

- The function definition of the interrupt function
- The argument list
- The value of :LEVEL (an integer in the range 0 through 7)
- The value of :ONCE-ONLY-P (T or NIL)

If the interrupt function represented by *iif-id* has been uninstated, GET-INTERRUPT-FUNCTION returns four values of NIL.

Example

```
Lisp> (defun time-elapsed (n)
         (format t
                 "~@(~R~) second~:P ~:*~[have~;has~:;have~] ~
                  elapsed since setting the timer"
                 n))
TIME-ELAPSED
Lisp> (setf t-e-iif (instate-interrupt-function
                      #'time-elapsed
                      :arguments (list 5)))
8388671
Lisp> (get-interrupt-function t-e-iif)
#<Interpreted Function (LAMBDA (N) (BLOCK TIME-ELAPSED (FORMAT T
"~@(~R~) second~:P ~:*~[have~;has~:;have~] ~
                  elapsed since setting the timer" N))) 4742880> ;
(5) ;
2 ;
NIL
```

- The function TIME-ELAPSED, which prints out the number of seconds since a timer was set, is defined. It takes a single argument.
- TIME-ELAPSED is instated as an interrupt function. The :ARGUMENTS keyword specifies that TIME-ELAPSED is passed one argument, the number 5. The *iif-id* returned by INSTATE-INTERRUPT-FUNCTION is retained as the value of T-E-IIF.
- The call to GET-INTERRUPT-FUNCTION returns four values. The first value is the function definition of TIME-ELAPSED. The second value is a list of the arguments specified with INSTATE-INTERRUPT-FUNCTION. The third value is the interrupt level (2, the default for INSTATE-INTERRUPT-FUNCTION). The fourth value is NIL, indicating that :ONCE-ONLY-P was not specified with INSTATE-INTERRUPT-FUNCTION.

GET-KEYBOARD-FUNCTION Function

Returns information about the function that is bound to a control character.

Format

GET-KEYBOARD-FUNCTION *control-character*

Argument

control-character
The control character to which a function is bound.

Return Values

Three values:

- The function that is bound to the control character
- The function's argument list
- The function's interrupt level

If no function is bound to the specified control character, the GET-KEYBOARD-FUNCTION returns NIL for all three values.

Examples

1.
```
Lisp> (bind-keyboard-function #\^B #'break)
T
Lisp> (get-keyboard-function #\^B)
#<Compiled Function BREAK #x261510> ;
NIL ;
1
```

   - The call to the BIND-KEYBOARD-FUNCTION function binds Ctrl/B to the BREAK function.
   - The call to the GET-KEYBOARD-FUNCTION function returns the function to which Ctrl/B is bound; the function's argument list (which is NIL); and the function's interrupt level (which is 1).

2.
```
Lisp> (get-keyboard-function #\^S)
NIL ;
NIL ;
NIL
```

   All three values returned are NIL, because Ctrl/S is not bound to any function.

GET-PROCESS-INFORMATION Function

Returns information about a process. If the process is nonexistent, this function returns NIL. The keywords you specify with the function determine the type of information the function returns.

This function is similar to the VMS system service $GETJPI. For more information on the $GETJPI system service, see the *VMS System Services Reference Manual.*

Format

GET-PROCESS-INFORMATION *process* **&REST** {*keyword*}*

Arguments

process
The name or the identification of the process (PID) about which information is to be returned. You can specify a string, an integer, or NIL. If you specify a string, the argument is the process name; if you specify an integer, the argument is the PID; if you specify NIL, the information the function returns corresponds to the current process.

keyword
Optional keywords that return specific types of information about the process. Do not specify values with the keywords.

Table 8 lists the keywords that you can specify and the values they return.

Table 8: GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:ACCOUNT	A string that specifies the account.
:ACTIVE-PAGE-TABLE-COUNT	An integer that specifies the active page table count.
:AST-ACTIVE	A vector of four bits that specifies the number of access modes that have active asynchronous system traps (ASTs) for the process.
:AST-COUNT	An integer that specifies the remaining AST quota.
:AST-ENABLED	A vector of four bits that specifies the number of access modes that have enabled ASTs for the process.
:AST-QUOTA	An integer that specifies the AST quota.
:AUTHORIZED-PRIVILEGES	A vector of 64 bits that specifies the privileges the process is authorized to enable.
:BASE-PRIORITY	An integer that specifies the base priority.
:BATCH	Either T or NIL. The function returns T if the process is a batch job; otherwise, returns NIL.
:BIO-BYTE-COUNT	An integer that specifies the remaining buffered I/O byte count quota.
:BIO-BYTE-QUOTA	An integer that specifies the buffered I/O byte count quota.
:BIO-COUNT	An integer that specifies the remaining buffered I/O operation quota.
:BIO-OPERATIONS	An integer that specifies the number of buffered I/O operations the process has performed.
:BIO-QUOTA	An integer that specifies the buffered I/O operation quota.
:CLI-TABLENAME	A string that specifies the file name of the current command language interpreter table.
:CPU-LIMIT	An integer that specifies the CPU time limit of the process in 10-millisecond units.

(continued on next page)

Table 8 (Cont.): GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:CPU-TIME	An integer that specifies the accumulated CPU time of the process in 10-millisecond units.
:CURRENT-PRIORITY	An integer that specifies the current priority.
:CURRENT-PRIVILEGES	A vector of 64 bits that specifies the current privileges.
:DEFAULT-PAGE-FAULT-CLUSTER	An integer that specifies the default page fault cluster size.
:DEFAULT-PRIVILEGES	A vector of 64 bits that specifies the default privileges.
:DIO-COUNT	An integer that specifies the remaining direct I/O operation quota.
:DIO-OPERATIONS	An integer that specifies the number of direct I/O operations the process has performed.
:DIO-QUOTA	An integer that specifies the direct I/O operation quota.
:ENQUEUE-COUNT	An integer that specifies the number of lock manager enqueues.
:ENQUEUE-QUOTA	An integer that specifies the lock manager enqueue quota.
:EVENT-FLAG-WAIT-MASK	A vector of 32 bits that specifies the event flag wait mask.
:FIRST-FREE-P0-PAGE	An integer that specifies the first free page at the end of the program region.
:FIRST-FREE-P1-PAGE	An integer that specifies the first free page at the end of the control region.
:GLOBAL-PAGES	An integer that specifies the number of global pages in the working set.
:GROUP	An integer that specifies the group field of the UIC.
:IMAGE-NAME	A string that specifies the current image file name.
:IMAGE-PRIVILEGES	A vector of 64 bits that specifies the privileges with which the current image of the process was installed.
:JOB-SUBPROCESS-COUNT	An integer that specifies the number of subprocesses.
:LOCAL-EVENT-FLAGS	A vector of 32 bits that specifies the local event flags the process has in effect.
:LOGIN-TIME	An integer in internal time that specifies the time the process was created.
:MEMBER	An integer that specifies the member field of the UIC.
:MOUNTED-VOLUMES	An integer that specifies the number of mounted volumes.

(continued on next page)

Table 8 (Cont.): GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:OPEN-FILE-COUNT	An integer that specifies the remaining open file quota.
:OPEN-FILE-QUOTA	An integer that specifies the open file quota.
:OWNER-PID	An integer that specifies the process ID of the owner.
:PAGE-FAULTS	An integer that specifies the number of page faults.
:PAGE-FILE-COUNT	An integer that specifies the number of paging file pages remaining to the process.
:PAGE-FILE-QUOTA	An integer that specifies the paging file quota.
:PAGES-AVAILABLE	An integer that specifies the number of virtual pages available for expansion.
:PID	An integer that specifies the process ID.
:PID-OF-PARENT	An integer that specifies the PID of the parent process. This integer differs from :OWNER-PID in that :PID-OF-PARENT refers to the top-level process, while :OWNER-PID refers to the process immediately above the current process or subprocess.
:PROCESS-CREATION-FLAGS	A 32-bit bit-vector that specifies the flags used to create the process.
:PROCESS-INDEX	An integer that specifies the index number of the process at a given instant. (Process index numbers are reassigned to different processes over time.)
:PROCESS-NAME	A string that specifies the name of the process.
:SITE-SPECIFIC	A longword that specifies the contents of the site-specific longword.
:STATE	An integer that specifies the state of the process.
:STATUS	A vector of 32 bits that specifies the status flags.
:SUBPROCESS-COUNT	An integer that specifies the number of subprocesses owned by the process.
:SUBPROCESS-QUOTA	An integer that specifies the subprocess quota.
:TERMINAL	A string that specifies the name of the terminal with which the process is interacting.
:TERMINATION-MAILBOX	An integer that specifies the termination mailbox unit number.
:TIMER-QUEUE-COUNT	An integer that specifies the remaining timer queue entry quota.
:TIMER-QUEUE-QUOTA	An integer that specifies the timer queue entry quota.
:UAF-FLAGS	A 12-bit bit-vector that specifies the UAF flags of the user who owns the process.
:UIC	An integer that specifies the UIC.
:USERNAME	A string that specifies the user name.

(continued on next page)

Table 8 (Cont.): GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
`:VIRTUAL-ADDRESS-PEAK`	An integer that specifies the peak virtual address space size.
`:WORKING-SET-AUTHORIZED-EXTENT`	An integer that specifies the maximum authorized working set extent.
`:WORKING-SET-AUTHORIZED-QUOTA`	An integer that specifies the authorized working set quota.
`:WORKING-SET-COUNT`	An integer that specifies the number of process pages in the working set.
`:WORKING-SET-DEFAULT`	An integer that specifies the default working set size.
`:WORKING-SET-EXTENT`	An integer that specifies the current working set size extent.
`:WORKING-SET-PEAK`	An integer that specifies the peak working set size.
`:WORKING-SET-QUOTA`	An integer that specifies the current working set quota.
`:WORKING-SET-SIZE`	An integer that specifies the current working set size.

Return Value

The keywords and their values are returned as a list in the following format:

(*:keyword-1 value-1 :keyword-2 value-2* . . .)

The function preserves the order of the keyword-value pairs in the argument list.

If you do not specify keywords, the function returns a list of all the keyword-value pairs. If the specified process does not exist, the function returns `NIL`.

Examples

1.
```
Lisp> (get-process-information "smith"
                               :batch
                               :cpu-time
                               :base-priority
                               :global-pages)
(:BATCH NIL :CPU-TIME 45884 :BASE-PRIORITY 4 :GLOBAL-PAGES 68)
```

Returns the value of the batch setting, the CPU time, the base priority, and the number of global pages used for the process SMITH.

2.
```
Lisp> (defun parent nil
        (let ((pid
                (second (get-process-information
                         nil
                         :owner-pid))))
          (if (zerop pid) nil (attach pid))))
PARENT
```

GET-PROCESS-INFORMATION Function

Defines a function that just returns NIL if the LISP system is running in the main process, and attaches your process to the parent process if the system is running in a subprocess.

GET-TERMINAL-MODES Function

Returns information about the terminal characteristics of the device associated with the *TERMINAL-IO* variable when you invoke the LISP system. If the specified stream is not connected to a terminal, the LISP system signals an error. The keywords you specify with the function determine the type of information that the function returns.

This function is similar to the DCL SHOW TERMINAL command. For more information on the SHOW TERMINAL command, see the *VMS DCL Dictionary*.

Format

GET-TERMINAL-MODES &REST *keyword*

Argument

keyword

Optional keywords that return the terminal characteristics of the stream that is bound to the *TERMINAL-IO* variable. Do not specify values with the keywords.

Table 9 lists the keywords that you can specify and the values they return.

Table 9: GET-TERMINAL-MODES Keywords

Keyword	Return Value
:BROADCAST	Either T or NIL. The function returns T if your terminal can receive broadcast messages, such as MAIL notifications and REPLY messages; otherwise, returns NIL.
:ECHO	Either T or NIL. The function returns T if the terminal displays the input character that it receives; otherwise, returns NIL. If the function returns NIL, the terminal displays only data output from the system or a user application program.
:ESCAPE	Either T or NIL. The function returns T if ANSI standard escape sequences transmitted from the terminal are handled as a single multicharacter terminator; otherwise, returns NIL. The terminal driver checks the escape sequences for syntax before passing them to the program. For more information on escape sequences, see the *VMS I/O User's Reference Manual: Part I.*

(continued on next page)

Table 9 (Cont.): GET-TERMINAL-MODES Keywords

Keyword	Return Value
:HALF-DUPLEX	Either T or NIL. The function returns T if the terminal's operating mode is half-duplex, and the function returns NIL if the operating mode is full-duplex. For a description of terminal operating modes, see the *VMS I/O User's Reference Manual: Part I.*
:PASS-ALL	Either T or NIL. The function returns T if the system does not expand tab characters to blanks, fill carriage return or linefeed characters, recognize control characters, and receive broadcast messages. The function returns NIL if the system passes all data to an application program as binary data.
:PASS-THROUGH	Either T or NIL. This mode is the same as the :PASS-ALL mode, except that "TTSYNC" protocol (Ctrl/S and Ctrl/Q) is still used.
:TYPE-AHEAD	Either T or NIL. The function returns T if the terminal accepts input that is typed when there is no outstanding read, and the function returns NIL if the terminal driver is dedicated and accepts input only when a program or the system issues a read.
:WRAP	Either T or NIL. The function returns T if the terminal generates a carriage return and a line feed when the end of a line is reached. Otherwise, the function returns NIL. The end of the line is determined by the terminal-width setting.

NOTE

> :PASS-ALL has been kept for the sake of compatibility with Version 1 of VAX LISP, but it is not recommended that you use :PASS-ALL.

Return Value

The keywords and their values are returned as a list in the following format:

(*:keyword-1 value-1 :keyword-2 value-2* . . .)

The function preserves the order of the keyword-value pairs in the argument list.

If you do not specify keywords, the function returns a list of the keyword-value pairs. The list is returned in a format such that the list can be specified as an argument in a call to the SET-TERMINAL-MODES function.

Example

```
Lisp> (get-terminal-modes)
(:BROADCAST T :ECHO T :ESCAPE NIL :HALF-DUPLEX NIL :PASS-ALL NIL
:TYPE-AHEAD T :WRAP T :PASS-THROUGH NIL)
```

Returns a list of all the keyword-value pairs.

GET-VMS-MESSAGE Function

Returns the system message associated with a specified VMS status.

Format

GET-VMS-MESSAGE *status* **&OPTIONAL** *flags*

Arguments

status
A fixnum that specifies the VMS status code of the message that is to be returned. See the *VMS System Messages and Recovery Procedures Reference Manual* for information on VMS message status codes.

flags
A bit vector of length four that specifies the content of the message. The default value is `#*0000`, which indicates that the process default message flags are to be used. The information that is included in the message when each of the four bits is set follows:

Bit	Information
0	Text
1	Message ID
2	Severity
3	Facility

Return Value

Returns the message that corresponds to the specified status code as a string. The function returns `NIL` if you specify a status code that does not exist.

Examples

1.
```
Lisp> (get-vms-message 32)
"%SYSTEM-W-NOPRIV, no privilege for attempted operation"
```

   Returns the VMS message text for message 32 with all flags set.

2.
```
Lisp> (get-vms-message 32 #*1001)
"%SYSTEM, no privilege for attempted operation"
```

   Returns the VMS message text for message 32 with only the facility and text flags set.

HASH-TABLE-REHASH-SIZE Function

Returns the rehash size of a hash table. The rehash size indicates how much a hash table is to increase when it is full. You specify that value when you create a hash table with the MAKE-HASH-TABLE function. For information on hash tables, see *Common LISP: The Language.*

Format

HASH-TABLE-REHASH-SIZE *hash-table*

Argument

hash-table
The name of the hash table whose rehash size is to be returned.

Return Value

An integer greater than 0 or a floating-point number greater than 1. If an integer is returned, the value indicates the number of entries that are to be added to the table. If a floating-point number is returned, the value indicates the ratio of the new size to the old size.

Example

```
Lisp> (setf *print-array* nil)
NIL
Lisp> (setf table-1 (make-hash-table :test #'equal
                                     :size 200
                                     :rehash-size 1.5
                                     :rehash-threshold .95))
#<Hash Table #x503BA8>
Lisp> (hash-table-rehash-size table-1)
1.5
```

- The first call to the SETF macro sets the value of the *PRINT-ARRAY* variable to NIL.
- The second call to the SETF macro sets TABLE-1 to the hash table created by the call to the MAKE-HASH-TABLE function.
- The call to the HASH-TABLE-REHASH-SIZE function returns the rehash size of the hash table, TABLE-1.

HASH-TABLE-REHASH-THRESHOLD Function

Returns the rehash threshold for a hash table. The rehash threshold indicates how full a hash table can get before its size has to be increased. You specify that value when you create a hash table with the MAKE-HASH-TABLE function. For information on hash tables, see *Common LISP: The Language*.

Format

HASH-TABLE-REHASH-THRESHOLD *hash-table*

Argument

hash-table
The hash table whose rehash threshold is to be returned.

Return Value

An integer greater than 0 and less than hash table's rehash size or a floating-point number greater than 0 and less than 1.

Example

```
Lisp> (setf *print-array* nil)
NIL
Lisp> (setf table-1 (make-hash-table :test #'equal
                                     :size 200
                                     :rehash-size 1.5
                                     :rehash-threshold .95))
#<Hash Table #x503BA8>
Lisp> (hash-table-rehash-threshold table-1)
0.95
```

- The first call to the SETF macro sets the value of the *PRINT-ARRAY* variable to NIL.
- The second call to the SETF macro sets TABLE-1 to the hash table created by the call to the MAKE-HASH-TABLE function.
- The call to the HASH-TABLE-REHASH-THRESHOLD function returns the rehash threshold of the hash table, TABLE-1.

HASH-TABLE-SIZE Function

Returns the current size of a hash table. You specify that value when you create a hash table with the MAKE-HASH-TABLE function. For information on hash tables, see *Common LISP: The Language.*

Format

HASH-TABLE-SIZE *hash-table*

Argument

hash-table
The hash table whose initial size is to be returned.

Return Value

An integer that indicates the initial size of the hash table.

Example

```
Lisp> (setf *print-array* nil)
NIL
Lisp> (setf table-1 (make-hash-table :test #'equal
                                     :size 200
                                     :rehash-size 1.5
                                     :rehash-threshold .95))

#<Hash Table #x503BA8>
Lisp> (hash-table-size table-1)
233
```

- The first call to the SETF macro sets the value of the *PRINT-ARRAY* variable to NIL.
- The second call to the SETF macro sets TABLE-1 to the hash table created by the call to the MAKE-HASH-TABLE function.
- The call to the HASH-TABLE-SIZE function returns the initial size of the hash table, TABLE-1.

HASH-TABLE-TEST Function

Returns a symbol that indicates how a hash table's keys are compared. The value is specified when you create a hash table with the MAKE-HASH-TABLE function. For information on hash tables, see *Common LISP: The Language*.

Format

HASH-TABLE-TEST *hash-table*

Argument

hash-table
The hash table whose test value is to be returned.

Return Value

A symbol: (EQ, EQL, or EQUAL). EQL is the default when creating a hash table.

Example

```
Lisp> (setf *print-array* nil)
NIL
Lisp> (setf table-1 (make-hash-table :test #'equal
                                     :size 200
                                     :rehash-size 1.5
                                     :rehash-threshold .95))
#<Hash Table #x503BA8>
Lisp> (hash-table-test table-1)
EQUAL
```

- The first call to the SETF macro sets the value of the *PRINT-ARRAY* variable to NIL.
- The second call to the SETF macro sets TABLE-1 to the hash table created by the call to the MAKE-HASH-TABLE function.
- The call to the HASH-TABLE-TEST function returns the test for the hash table, TABLE-1.

IMMEDIATE-OUTPUT-P Function

Predicate indicates whether an output stream does not buffer its output. The VAX LISP I/O system uses this function to improve output performance by buffering output when the stream itself does not perform buffering.

Format

IMMEDIATE-OUTPUT-P &OPTIONAL *output-stream*

Argument

output-stream
An output stream. The default value is `*STANDARD-OUTPUT*`. If you supply a value of `T`, the value of `*TERMINAL-IO*` is used.

Return Value

`T`, if *output-stream* does not buffer output; otherwise, `NIL`.

INSPECT Function

Invokes the VAX LISP Inspector, a utility for examining and modifying objects in your current LISP environment. The Inspector displays the components of the LISP object you specify. You can inspect these components in turn and modify their values.

NOTE

> The VAX LISP Inspector is available only when LISP is running with the DECwindows-based development environment.

You can run the Inspector either synchronously or asynchronously (the default). In synchronous mode, you can specify which value the Inspector is to return. In asynchronous mode, the Inspector immediately returns the object on which it was invoked to the program (or other VAX LISP utility) from which it was invoked.

See Chapter 9 of the *VAX LISP/VMS Program Development Guide* for more information on using the Inspector.

Format

INSPECT &OPTIONAL *object* **&KEY :PARALLEL**

Arguments

object
Any LISP object.

:PARALLEL
Specifies whether the Inspector runs asynchronously (`:PARALLEL T`) or synchronously (`:PARALLEL NIL`) with other programs in your LISP environment. The default value is `T`.

Return Value

The returned value depends on the mode of operation. If the Inspector is running asynchronously, it immediately returns the object on which it was invoked. If the Inspector is running synchronously, there are two ways to return a value:

- You may specify an object whose value the Inspector will return by selecting that object and choosing the Return item from the Operations menu.
- Otherwise, the Inspector returns the object on which it was invoked when you exit the Inspector.

INSTATE-INTERRUPT-FUNCTION Function

Takes as its first argument a function that will later be invoked asynchronously and returns an identification (*iif-id*) for this instance of the function. The *iif-id* is intended to be passed to a routine that can cause an AST. When the AST occurs, it invokes the interrupt function identified by the *iif-id*.

The `:ARGUMENTS` keyword allows you to supply a list of zero or more arguments that are passed to the interrupt function when it executes. This allows a single function to take different actions, depending on the particular AST that invokes it.

The `:LEVEL` keyword lets you specify the interrupt level for the interrupt function as an integer in the range 0 through 7. See Chapter 7 in the *VAX LISP/VMS System Access Guide* for more information about interrupt levels.

The `:ONCE-ONLY-P` keyword allows you to specify that this instance of the function will be invoked only once and then discarded. Specifying `:ONCE-ONLY-P T` is equivalent to using `UNINSTATE-INTERRUPT-FUNCTION` on the function after its first invocation. However, `:ONCE-ONLY-P` does not disable further occurrences of the AST after its first occurrence. If `:ONCE-ONLY-P T` is specified and the corresponding AST occurs more than once, the second and subsequent ASTs are ignored. (See `UNINSTATE-INTERRUPT-FUNCTION` for more details.)

For more information about interrupt functions, see Chapter 6 in the *VAX LISP/VMS System Access Guide.*

INSTATE-INTERRUPT-FUNCTION Function

Format

INSTATE-INTERRUPT-FUNCTION *function*
&KEY :ARGUMENTS :LEVEL :ONCE-ONLY-P

Arguments

function
A function to be invoked asynchronously at a later time.

:ARGUMENTS
A list of zero or more arguments to be passed to the interrupt function when it is invoked.

:LEVEL
An integer in the range 0 through 7, specifying the interrupt level for the interrupt function. The default interrupt level is 2.

:ONCE-ONLY-P
T or NIL (the default), specifying whether or not this instance of the function is to be uninstated when it has been invoked once.

Return Value

An integer that identifies this instance of the interrupt function. This integer becomes the *iif-id* argument to functions that require an *iif-id* and the *astprm* argument to external routines that can cause an AST.

Examples

1.
```
Lisp>  (define-external-routine (sys$setimr
                               :check-status-return t)
         (efn :mechanism :value)
         (daytim :vax-type :quadword)
         (astadr :mechanism :value)
         (astprm :mechanism :value))
SYS$SETIMR
Lisp>  (define-external-routine (sys$bintim
                               :check-status-return t)
         (timbuf :vax-type :text :lisp-type string)
         (timadr :vax-type :quadword :access :in-out))
SYS$BINTIM
Lisp>  (defun set-timer (delta-time)
         (let ((iif-id (instate-interrupt-function
                        #'timer-interrupt-handler
                        :once-only-p t)))
           (call-out sys$setimr nil delta-time
                               common-ast-address iif-id))
        t)
SET-TIMER
Lisp>  (defun timer-interrupt-handler ()
         (print "The timer has expired"))
TIMER-INTERRUPT-HANDLER
Lisp>  (setq delta 0)  ; delta must be bound before call-out
0
Lisp>  (call-out sys$bintim "0 ::5" delta)
1
Lisp>  (set-timer delta)
T
Lisp>  (five seconds pass) "The timer has expired"
```

- The external routine SYS$SETIMR is defined. SYS$SETIMR is a system service that sets a timer and causes an AST when the timer expires. The ASTADR and ASTPRM arguments are both passed with :MECHANISM :VALUE.
- The external routine SYS$BINTIM is defined. SYS$BINTIM is a system service that converts a time specified as a string to a binary format acceptable to SYS$SETIMR.
- The function SET-TIMER is defined. SET-TIMER's argument is the binary-formatted time before a timer should expire. SET-TIMER calls INSTATE-INTERRUPT-FUNCTION to instate TIMER-INTERRUPT-HANDLER as an interrupt function. The T value for :ONCE-ONLY-P requests that the interrupt function be uninstated after it executes once. SET-TIMER then calls out to SYS$SETIMR, passing the binary time as the second argument. The third argument is (and must be) the COMMON-AST-ADDRESS parameter; the fourth argument is the *iif-id* returned by INSTATE-INTERRUPT-FUNCTION.
- The function TIMER-INTERRUPT-HANDLER is defined. It simply prints a message on the terminal.
- After the binary format for 5 seconds is stored in DELTA, the call to SET-TIMER sets a timer to expire in 5 seconds. SET-TIMER returns. Five seconds later, the timer expires and the interrupt function TIMER-INTERRUPT-HANDLER executes, printing the message.

2.

```
Lisp> (defun set-timer (seconds)
        (let ((delta 0)
              (iif (instate-interrupt-function
                       #'time-elapsed
                       :once-only-p t
                       :arguments (list seconds))))
          (call-out sys$bintim (time-string seconds) delta)
          (call-out sys$setimr nil delta
                               common-ast-address iif))
        t)
SET-TIMER
Lisp> (defun time-string (n)
         (format nil "0 :~d:~d" (truncate n 60) (mod n 60)))
TIME-STRING
Lisp> (defun time-elapsed (n)
        (format t
                "~@(~R~) second~:P ~:*~[have~;has~:;have~] ~
                   elapsed since setting the timer"
                n))
TIME-ELAPSED
Lisp> (set-timer 5)
T
Lisp> (five seconds elapse) Five seconds have elapsed since
setting the timer
```

This example shows the use of arguments with interrupt functions. The external routines SYS$SETIMR and SYS$BINTIM have the same definitions as shown in Example 1.

- The new definition of SET-TIMER accepts an integer argument that is the number of seconds to wait (not a binary-formatted time). SET-TIMER instates a function called TIME-ELAPSED as an interrupt function, requesting that one argument (the number of seconds) be passed to TIME-ELAPSED. SET-TIMER then calls out to SYS$BINTIM to convert the seconds to binary format. (An auxiliary function, TIME-STRING, converts the integer argument to a string acceptable to SYS$BINTIM. TIME-STRING cannot format an argument larger than 3599 seconds properly.) Finally, SET-TIMER calls out to SYS$SETIMR, passing the binary-formatted time (the second argument) and the *iif-id* for TIME-ELAPSED (the fourth argument).

- The function TIME-ELAPSED is defined. It accepts an integer argument and uses FORMAT to print the number of seconds represented by that argument.

- SET-TIMER is called with the argument 5. SET-TIMER returns. After 5 seconds elapse, TIME-ELAPSED executes and prints the formatted message on the terminal, including the number of seconds.

3.

```
Lisp> (defun print-button (button transition)
        (when transition
          (case button
            (#.uis:pointer-button-1
                (princ "Left button pressed"))
            (#.uis:pointer-button-2
                (princ "Middle button pressed"))
            (#.uis:pointer-button-3
                (princ "Right button pressed")))))
PRINT-BUTTON
Lisp> (setf button-iif
            (instate-interrupt-function #'print-button))
8454171
Lisp> (uis:set-button-action display window button-iif)
T
Lisp>
```

This example shows the use of an interrupt function with a VAX LISP-supplied function. This example works only on a VAXstation running UIS.

- The function PRINT-BUTTON is defined. Depending on its arguments, it prints one of three lines on the terminal, or it does nothing.
- PRINT-BUTTONS is instated as an interrupt function. The *iif-id* returned by INSTATE-INTERRUPT-FUNCTION is retained as the value of BUTTON-IIF.
- The function SET-BUTTON-ACTION is called with BUTTON-IIF as the third argument. SET-BUTTON-ACTION specifies what should happen when a workstation pointer button is pressed or released while the pointer cursor is in a specified window. If an *iif-id* is passed as the third argument, the associated interrupt function is invoked when a button is pressed or released. SET-BUTTON-ACTION causes an interrupt function to be passed two arguments: the button involved, and T or NIL to indicate whether the button was pressed or released.
- After SET-BUTTON-ACTION returns, a button is pressed. PRINT-BUTTON receives the two arguments passed to it and prints the message on the screen.

LINE-POSITION Function

Returns the number of characters that have been output on the current line, if that number can be determined; otherwise, NIL.

Format

LINE-POSITION &OPTIONAL *output-stream*

Argument

output-stream
An output stream. The default value is *STANDARD-OUTPUT*. If you specify T, the value of *TERMINAL-IO* is used.

Return Value

A fixnum or NIL.

LISTEN2 Function

Returns two values instead of the one returned by the Common LISP LISTEN function, enabling you to find out if end-of-file was encountered on the input stream. You can use this function wherever you would normally use LISTEN.

Format

LISTEN2 &OPTIONAL *input-stream*

Arguments

input-stream
An input stream. The default value is *STANDARD-INPUT*. If you supply a value of T, the value of *TERMINAL-IO* is used.

Return Values

Two values:

- T, if a character is immediately available from *input-stream*; otherwise, NIL.
- T, if end-of-file was encountered on *input-stream*; otherwise, NIL.

LOAD Function

Reads and evaluates the contents of a file into the LISP environment.

In VAX LISP, if the specified file name does not specify an explicit file type, the LOAD function locates the source file (type .LSP) or fast-loading file (type .FAS) with the latest file write date and loads it. This ensures that the latest version of the file is loaded, whether or not the file is compiled.

Format

LOAD *filename*
&KEY :IF-DOES-NOT-EXIST :PRINT :VERBOSE

Arguments

filename
The name of the file to be loaded.

:IF-DOES-NOT-EXIST

Specifies whether the LOAD function signals an error if the file does not exist. The value can be T or NIL. If you specify T, the function signals an error if the file does not exist. If you specify NIL, the function returns NIL if the file does not exist. The default value is T.

:PRINT

Specifies whether the value of each form that is loaded is printed to the stream bound to the *STANDARD-OUTPUT* variable. The value can be T or NIL. If you specify T, the value of each form in the file is printed to the stream. If you specify NIL, no action is taken. The default value is NIL. This keyword is useful for debugging.

:VERBOSE

Specifies whether the LOAD function is to print a message in the form of a comment to the stream bound to the *STANDARD-OUTPUT* variable. The value can be T or NIL. If you specify T, the function prints a message. The message includes information such as the name of the file that is being loaded. If you specify NIL, the function uses the value of *LOAD-VERBOSE* variable. The default is T.

Return Value

A value other than NIL if the load operation is successful.

Example

```
Lisp> (compile-file "factorial")

Starting compilation of file DBA1:[SMITH]FACTORIAL.LSP;1

FACTORIAL compiled.

Finished compilation of file DBA1:[SMITH]FACTORIAL.LSP;1
0 Errors, 0 Warnings
"DBA1:[SMITH]FACTORIAL.FAS;1"
Lisp> (load "factorial")
; Loading contents of file DBA1:[SMITH]FACTORIAL.FAS;1
;   FACTORIAL
; Finished loading DBA1:[SMITH]FACTORIAL.FAS;1
T
```

- The call to the COMPILE-FILE function produces a fast-loading file named FACTORIAL.FAS.

- The call to the LOAD function locates the fast-loading file FACTORIAL.FAS and loads the file into the LISP environment.

LONG-SITE-NAME Function

Translates the logical name LISP$LONG_SITE_NAME. If the first character of the resulting string is an at sign (@), the rest of the string is assumed to be a file specification. The file is read and its content is returned as a string that represents the physical location of the computer hardware on which the VAX LISP system is running. If the first character of the translation is not an at sign, the translation itself is returned as the long-site name.

Format

LONG-SITE-NAME

Argument

None.

Return Value

The contents of a file or the translation of the logical name LISP$LONG_SITE_NAME is returned as a string that represents the physical location of the computer hardware on which the VAX LISP system is running. If a long-site name is not defined, `NIL` is returned.

Example

```
Lisp> (long-site-name)
"Smith's Computer Company
Artificial Intelligence Group
22 Plum Road
Canterbury, Ohio 47190"
```

MACHINE-INSTANCE Function

Translates the logical name LISP$MACHINE_INSTANCE.

Format

MACHINE-INSTANCE

Argument

None.

Return Value

The translation of the logical name LISP$MACHINE_INSTANCE is returned as a string. If the logical name is not defined and DECnet–VAX is running, the node name is returned. If the logical name is not defined and DECnet–VAX is not running, `NIL` is returned.

Example

```
Lisp> (machine-instance)
"MIAMI"
```

MACHINE-VERSION Function

Returns the content of the system identification (SID) register as a string that represents the version of computer hardware on which the VAX LISP system is running. The contents of the SID register are determined by the type of CPU—for example, 780, 750, or 730. For more information about CPU types, see the *VAX Architecture Handbook*.

Format

MACHINE-VERSION

Argument

None.

Return Value

The contents of the SID register are returned as a string.

Example

```
Lisp> (machine-version)
"SID Register: #x01383550"
```

MAKE-ARRAY Function

Creates and returns an array. VAX LISP has added the :ALLOCATION keyword to this Common LISP function. When the function is used with the :ALLOCATION keyword and the value :STATIC, the function creates a statically allocated array.

During system usage, the garbage collector moves LISP objects. You can prevent the garbage collector from moving an object by allocating it in static space. Arrays, vectors, and strings can be statically allocated if you use the :ALLOCATION keyword and :STATIC value in a call to the MAKE-ARRAY function. Once an object is statically allocated, its virtual address does not change. Note that such objects are never garbage collected and their space cannot be reclaimed. By default, LISP objects are allocated in dynamic space.

NOTE

> A statically allocated object maintains its memory address even if a SUSPEND/RESUME operation is performed.

Calling the MAKE-ARRAY function with the :ALLOCATION :STATIC keyword-value pair is useful if you are creating a large array. Preventing the garbage collector from moving the array causes the garbage collector to go faster.

The MAKE-ARRAY function has a number of other keywords that can be used. See *Common LISP: The Language* for information on the other MAKE-ARRAY keywords.

VAX LISP creates a specialized array when the array's element type is any of the types in Table 10.

Table 10: Specialized Array Element Types

CHARACTER	BIT	
(UNSIGNED-BYTE 2)	(UNSIGNED-BYTE 4)	(UNSIGNED-BYTE 8)
(UNSIGNED-BYTE 12)	(UNSIGNED-BYTE 16)	(UNSIGNED-BYTE 24)
(UNSIGNED-BYTE 32)	(UNSIGNED-BYTE 64)	
(SIGNED-BYTE 8)	(SIGNED-BYTE 16)	(SIGNED-BYTE 32)
(SIGNED-BYTE 64)		
SINGLE-FLOAT	DOUBLE-FLOAT	LONG-FLOAT

For subtypes of these types, VAX LISP creates a specialized array of the most specific type possible. For example:

```
Lisp> (type-of (make-arry 3 :element-type '(signed-byte 5)))
(SIMPLE-ARRAY (SIGNED-BYTE 8) (3))
```

For all other element types, VAX LISP creates a generalized array, with the element type T. For compatibility of VAX types with LISP types in calls to external routines, see the table on data conversion in Chapter 4 of the *VAX LISP/VMS System Access Guide*.

MAKE-ARRAY Function

Format

MAKE-ARRAY *dimensions*
&KEY :ALLOCATION *other-keywords*

Arguments

dimensions
A list of positive integers that are to be the dimensions of the array.

:ALLOCATION
Specifies whether the LISP object is to be statically allocated. You can specify one of the following values with the :ALLOCATION keyword:

:DYNAMIC	The LISP object is *not* to be statically allocated. This is the default.
:STATIC	The LISP object is to be statically allocated.

other-keywords
See *Common LISP: The Language.*

Return Value

The statically allocated object.

Example

```
Lisp> (defparameter bit-buffer
        (make-array '(1000 1000) :element-type 'bit
                                 :allocation :static))
BIT-BUFFER
```

Creates a large array of bits named BIT-BUFFER, which is not intended to be removed from the system. The :ELEMENT-TYPE keyword is one of the other keywords (described in *Common LISP: The Language*) that this function accepts.

MAKE-CALL-BACK-ROUTINE Function

Returns an alien structure of type CALL-BACK-ROUTINE, which can be passed to an external routine during callout. Chapter 4 in the *VAX LISP/VMS System Access Guide* contains more information on the callback facility.

Format

MAKE-CALL-BACK-ROUTINE *function*
&KEY :ARGUMENTS *argument-specifier*
:RESULT *result-specifier*

Arguments

function

Specifies the LISP function that will be called by an external routine. This argument may be a function object or a symbol that names a function. Symbols are useful if the named function is later redefined, or if you have not defined the function before the call to `MAKE-CALL-BACK-ROUTINE`.

:ARGUMENTS *argument-specifier*

Specifies the arguments to the callback routine. The *argument-specifier* can be one of the following:

- `NIL` indicates that the callback routine takes no arguments. This is the default.
- `:AP` indicates that the actual VAX argument list is passed to the callback routine as the only parameter. All arguments in the list must be accessed by the callback routine using alien field references. When the callback routine is invoked, it is passed a single argument that is an alien structure representing the VAX call frame argument list.
- A list of argument descriptions having either the format:

  (*argument-name*)

  or

  (*argument-name keyword-1 value-1*
  ...)
  ...)

  The *argument-name* must be a symbol. You may use the following keyword–value pairs:

  `:ACCESS` *value*

  Specifies the type of access to use when passing an argument. The *value* can be either `:IN` or `:IN-OUT`. Use `:IN-OUT` when the callback routine returns multiple values. The default value is `:IN`.

  `:MECHANISM` *value*

  Specifies the argument-passing mechanism used in passing data to and from the callback routine. The *value* can be `:VALUE`, `:REFERENCE`, or `:DESCRIPTOR`.

  The default argument-passing mechanism for arguments to a callback routine is `:DESCRIPTOR` when the `:VAX-TYPE` is `:TEXT`. The default mechanism for all other LISP data types is `:REFERENCE`.

  The default argument-passing mechanism for values returned from a callback routine is `:VALUE` for all scalar VAX data types except `:H-FLOATING`. The default mechanism for VAX type `:H-FLOATING` and all nonscalar types is `:REFERENCE`.

  `:LISP-TYPE` *type*

  Specifies the LISP type of arguments or return values. For arguments, the default `:LISP-TYPE` is `INTEGER`. When no `:VAX-TYPE` option is given, the default for a given `:LISP-TYPE` is used. Table 4–2 of your *VAX LISP/VMS System Access Guide* lists the default LISP type—VAX type pairings.

The default :LISP-TYPE for return values is INTEGER; the default :VAX-TYPE is :SIGNED-LONGWORD.

:VAX-TYPE *type*

Specifies the VAX type of arguments and return values. When no :VAX-TYPE option is given, the default value of its corresponding :LISP-TYPE is used. Table 4–2 of your *VAX LISP/VMS System Access Guide* lists default LISP type—VAX type pairings.

All :VAX-TYPE *type* values are keywords.

:RESULT *result-specifier*

Specifies the type of the value returned by the callback routine and conversion mechanisms from LISP to VAX data types. The default value is NIL, which means that the function returns no value. If it returns multiple values, the result must be the first value in the VALUES list. Subsequent arguments are processed in the order in which they are defined. The types of the returned values must match the argument's :LISP-TYPE.

The syntax for defining a *result-specifier* is similar to that for defining arguments. However, for a *result-specifier* you supply only the :VAX-TYPE and :LISP-TYPE options; :ACCESS and :MECHANISM keywords do not apply.

Return Value

An alien structure of type CALL-BACK-ROUTINE.

Examples

1.
```
(defvar my-call-back (make-call-back-routine
             #'integer-call-back
             :arguments
                 '((arg1 :lisp-type integer
                     :access :in
                     :mechanism :value
                     :vax-type :unsigned-longword)
                   (arg2 :lisp-type integer
                     :access :in-out
                     :mechanism :reference))
             :result
                 '(:lisp-type integer)))
```

MAKE-CALL-BACK-ROUTINE defines the name of the callback function (INTEGER-CALL-BACK) and the order of the arguments, as well as information about type and access characteristics. The second argument is defined to have :IN-OUT access; therefore, the callback routine will return multiple values.

2.
```
(let* ((lisp-call-back-routine
                 (make-call-back-routine
                     'integer-by-ap
                     :arguments :ap
                     :result '(:lisp-type integer)))
       .
       .
       .))
```

In this example, the callback function INTEGER-BY-AP is defined with the :AP keyword. Thus, it takes a VAX argument list as its only argument.

MEMORY-ALLOCATION-EXTENT Function

Returns the "allocation extent" currently used by the memory management system. The allocation extent is the minimum number of 64K-byte segments requested when it is necessary to enlarge LISP memory for internal reasons. The actual number of requested segments may exceed the allocation extent, depending on how much memory is required.

The allocation extent may be changed with the SETF macro. The new value must be a positive integer, representing a number of 64K-byte segments.

Format

MEMORY-ALLOCATION-EXTENT

Argument

None.

Return Value

An integer.

MODULE-DIRECTORY Variable

A variable whose value refers to the directory containing the module that is being loaded into the LISP environment due to a call to the REQUIRE function. The value is a pathname.

This variable is useful to determine the location of a module if additional files from the same directory must be loaded by the module. For example, consider the following contents of a file called REQUIRED_FILE1.LSP:

```
(provide "required_file1")
(load (merge-pathnames "required_file2" *module-directory*))
(defun test
  ...)
```

When you specify the preceding module with the REQUIRE function, you do not have to identify the module's directory if it is in one of the places the REQUIRE function searches (see the description of the REQUIRE function later in this manual). Furthermore, using the *MODULE-DIRECTORY* variable, as in this example, ensures that the file REQUIRED_FILE2 will be loaded from the same directory. After the module is loaded, the *MODULE-DIRECTORY* variable is rebound to NIL.

NOTE

As this variable is bound during calls to the REQUIRE function, nested calls to the function cause its value to be updated appropriately.

NREAD-LINE Function

NREAD-LINE, a destructive version of the Common LISP READ-LINE function, places the characters that were read into the string supplied as its first argument. NREAD-LINE returns the number of characters that were read, a flag indicating whether end-of-file was encountered, and a string containing the line if the line could not fit into the supplied string.

Format

NREAD-LINE *string* **&OPTIONAL** *input-stream eof-error-p eof-value-p recursive-p*

Arguments

string
A character string. NREAD-LINE updates *string* with the line that was read. If *string* has a fill pointer, the fill pointer is adjusted so that *string* appears to contain exactly what was read from the stream. If *string* is adjustable and the size of the line exceeds the size of *string*, then *string* is extended.

Since NREAD-LINE does not return *string*, you must maintain a pointer to *string*.

input-stream eof-error-p eof-value-p recursive-p
These arguments correspond to the arguments to READ-LINE, which is documented in *Common LISP: The Language*.

Return Values

Three values:

- A fixnum indicating the number of characters that were in the line.
- T, if the line was terminated by end-of-file; otherwise NIL.
- NIL, if the line fit into *string*: otherwise, a string containing the line.

OPEN-STREAM-P Function

This predicate indicates whether a stream is open.

Format

OPEN-STREAM-P *stream*

Argument

stream
A stream.

Return Value

T, if *stream* is open; NIL, if it is closed.

POST-GC-MESSAGE Variable

Controls the message that the LISP system displays after a garbage collection occurs. The value of this variable can be NIL, a string of message text, or the null string (""). If the value is NIL, the system displays a system message; if the value is a string, the system displays the string; if the variable's value is the null string (""), the system displays no output. The default value is NIL.

The system message is:

```
; ... Full GC finished
```

If you set the *POST-GC-MESSAGE* variable, the message you establish supersedes the system message displayed after a garbage collection.

Example

```
Lisp> (gc)
; Starting full GC ...
; ... Full GC finished
T
Lisp> (setf *post-gc-message* "")
""
Lisp> (gc)
; Starting full GC ...
T
Lisp> (setf *post-gc-message* "GC -- finished")
"GC -- finished"
Lisp> (gc)
; Starting full GC ...
GC -- finished
T
```

- The first call to the GC function shows the garbage collection messages that the LISP system displays by default.
- The first call to the SETF macro sets the value of the *POST-GC-MESSAGE* variable to the null string ("").
- The second call to the GC function shows that, if the variable's value is the null string, the system does not display a message when a garbage collection is finished.

- The second call to the SETF macro sets the value of the variable to the string "GC -- finished".
- The third call to the GC function shows that, if the variable's value is a string, the system displays the new message when a garbage collection is finished.

PPRINT-DEFINITION Function

Pretty-prints to a stream the function definition of a symbol.

Format

PPRINT-DEFINITION *symbol* **&OPTIONAL** *stream*

Arguments

symbol
The symbol whose function value is to be pretty-printed.

stream
The stream to which the code is to be pretty-printed. The default stream is the stream bound to the *STANDARD-OUTPUT* variable.

Return Value

No value.

Examples

1.
```
Lisp> (defun factorial (n)
"Returns the factorial of an integer."
(cond ((<= n 1) 1) (t (* n (factorial (- n 1))))))
FACTORIAL
Lisp> (pprint-definition 'factorial)
(DEFUN FACTORIAL (N)
  "Returns the factorial of an integer."
  (COND ((<= N 1) 1) (T (* N (FACTORIAL (- N 1))))))
```

   - The call to the DEFUN macro defines a function called FACTORIAL, which returns the factorial of an integer.
   - The call to the PPRINT-DEFINITION function pretty-prints the function value of the symbol FACTORIAL.

2.
```
Lisp> (defun record-my-statistics
(name age siblings married?)
(unless (symbolp name)
(error "~S must be a symbol." name))
(setf (get name 'age) age
(get name 'number-of-siblings) siblings
(get name 'is-this-person-married?) married?) name)
RECORD-MY-STATISTICS
Lisp> (pprint-definition 'record-my-statistics)
(DEFUN RECORD-MY-STATISTICS (NAME AGE SIBLINGS MARRIED?)
  (UNLESS (SYMBOLP NAME)
          (ERROR "~S must be a symbol." NAME))
  (SETF (GET NAME 'AGE) AGE
        (GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
        (GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED?)
  NAME)
```

- The call to the DEFUN macro defines a function called RECORD-MY-STATISTICS.
- The call to the PPRINT-DEFINITION function pretty-prints the function value of the symbol RECORD-MY-STATISTICS.

PPRINT-PLIST Function

Pretty-prints to a stream the property list of a symbol. A property list is a list of symbol-value pairs; each symbol is associated with a value or an expression. The PPRINT-PLIST function prints the property list in a way that emphasizes the relationship between the symbols and their values.

PPRINT-PLIST prints only the symbol-value pairs for which a symbol is accessible in the current package. (For information on packages, see *Common LISP: The Language*.) On the other hand, SYMBOL-PLIST returns all the symbol-value pairs (the entire property list) of a symbol, even those not accessible in the current package. So, the form (pprint-plist 'me) is not equivalent to the form (pprint (symbol-plist 'me)). The following example shows the differences between the two forms:

```
Lisp> (make-package 'planet)
Lisp> (setf (symbol-plist 'me)
            '(girl "samantha" boy "daniel"
              planet::inhabitant-of "earth"))
(GIRL "SAMANTHA" BOY "DANIEL" PLANET::INHABITANT-OF "EARTH")
Lisp> (pprint (symbol-plist 'me))
(GIRL "SAMANTHA" BOY "DANIEL" PLANET::INHABITANT-OF "EARTH")
Lisp> (pprint-plist 'me)
(GIRL "SAMANTHA"
 BOY "DANIEL")
```

The call to PPRINT prints the symbol-value pair PLANET::INHABITANT-OF "EARTH", but the call to PPRINT-PLIST does not. This is because the symbol INHABITANT-OF in the package PLANET is not accessible in the current package. (A symbol can be in another package and still be accessible in the current package.) The symbol ME in the current package is associated with the symbol-value pair INHABITANT-OF "EARTH" in the PLANET package, but the PPRINT-PLIST function does not print that symbol-value pair because it is not accessible in the current package.

PPRINT-PLIST Function

Format

PPRINT-PLIST *symbol* **&OPTIONAL** *stream*

Arguments

symbol
The symbol whose property list is to be pretty-printed.

stream
The stream to which the pretty-printed output is to be sent. The default stream is the stream bound to the *STANDARD-OUTPUT* variable.

Return Value

No value.

Examples

1.
```
Lisp> (setf (get 'children 'sons) '(danny geoffrey))
(DANNY GEOFFREY)
Lisp> (setf (get 'children 'daughters) 'samantha)
SAMANTHA
Lisp> (pprint-plist 'children)
(DAUGHTERS SAMANTHA
 SONS (DANNY GEOFFREY))
```

- The calls to the SETF macro give the symbol CHILDREN the properties SONS and DAUGHTERS. The property list of the symbol CHILDREN has two properties: DAUGHTERS whose value is SAMANTHA and SONS whose value is the list (DANNY GEOFFREY).
- The call to the PPRINT-PLIST function pretty-prints the property list of the symbol CHILDREN. The pretty-printed output emphasizes the relationship between each property and its value.

2.
```
Lisp> (defun record-my-statistics (name age siblings married?)
        (unless (symbolp name)
                (error "~S must be a symbol." name))
        (setf (get name 'age) age
              (get name 'number-of-siblings) siblings
              (get name 'is-this-person-married?) married)
        name)
RECORD-MY-STATISTICS
Lisp> (defun show-my-statistics (name)
        (unless (symbolp name)
                (error "~S must be a symbol." name))
                (pprint-plist name))
SHOW-MY-STATISTICS
Lisp> (record-my-statistics 'tom 29 3 nil)
TOM
Lisp> (show-my-statistics 'tom)
(IS-THIS-PERSON-MARRIED? NIL
 NUMBER-OF-SIBLINGS 3
 AGE 29)
```

- The first call to the DEFUN macro defines a function named RECORD-MY-STATISTICS.
- The second call to the DEFUN macro defines a function named SHOW-MY-STATISTICS. The definition includes a call to the PPRINT-PLIST function.
- The call to the RECORD-MY-STATISTICS function supplies the properties for the symbol TOM.
- The call to the SHOW-MY-STATISTICS function pretty-prints the property list for the symbol TOM.

PRE-GC-MESSAGE Variable

Controls the message the LISP system displays when a garbage collection starts. The value of this variable can be NIL, a string of message text, or the null string (""). If the value is NIL, the system displays a system message; if the value is a string of message text, the system displays the message text; if the variable's value is the null string, the system displays no output. The default value is NIL.

The system message is:

```
; Starting full GC ...
```

If you set the *PRE-GC-MESSAGE* variable, the message you establish supersedes the system message.

PRE-GC-MESSAGE Variable

Example

```
Lisp> (gc)
; Starting full GC ...
; ... Full GC finished
T
Lisp> (setf *pre-gc-message* "")
""
Lisp> (gc)
; ... Full GC finished
T
Lisp> (setf *pre-gc-message* "GC -- started")
"GC -- started"
Lisp> (gc)
GC -- started
; ... Full GC finished
T
```

- The first call to the GC function shows the garbage collection messages that are printed by default.
- The first call to the SETF macro sets the value of the *PRE-GC-MESSAGE* variable to the null string ("").
- The second call to the GC function causes the system not to display a message when the garbage collection starts.
- The second call to the SETF macro sets the value of the variable to the string "GC -- started".
- The third call to the GC function causes the system to display the new message text when the garbage collection starts.

PRINT-LINES Variable

Specifies the number of lines to be printed by an outermost logical block. The default for this variable is NIL, which specifies no abbreviation. *PRINT-LINES* is effective only when pretty-printing is enabled. When the system limits output to the number of lines specified by *PRINT-LINES*, it indicates abbreviation by replacing the last four characters on the last line printed with " . . . ".

The WRITE and WRITE-TO-STRING functions have been extended in VAX LISP to accept the :LINES keyword. If you specify this keyword, *PRINT-LINES* is bound to the value you supply with the keyword before any output is produced.

See *VAX LISP Implementation and Extensions to Common LISP* for more information on using the *PRINT-LINES* variable.

Example

```
Lisp> (setf *print-pretty* t)
T
Lisp> (setf *print-lines* 4)
4
Lisp> (format t "Stars: ~:!~/LINEAR/~."
'(polaris dubhe mira mirfak bellatrix capella algol
mirzam pollux canopus albireo castor alphecca
antares))
Stars: POLARIS
       DUBHE
       MIRA
       MIRFAK ...
```

With *PRINT-LINES* set to 4, printing stops at the end of the fourth line.

PRINT-MISER-WIDTH Variable

Controls miser mode printing. If the available line width between the indentation of the current logical block and the end of the line is less than the value of this variable, the pretty-printer enables miser mode. When output is printed in miser mode, all indentations are ignored. In addition, a new line is started for every conditional new line directive (~_, ~:_, or ~@_). The default value for *PRINT-MISER-WIDTH* is 40.

You can prevent the use of miser mode by setting the *PRINT-MISER-WIDTH* variable to NIL.

The WRITE and WRITE-TO-STRING functions have been extended in VAX LISP to accept the :MISER-WIDTH keyword. If you specify this keyword, *PRINT-MISER-WIDTH* is bound to the value you supply with the keyword before any output is produced.

For more information about miser mode and the use of the *PRINT-MISER-WIDTH* variable, see *VAX LISP Implementation and Extensions to Common LISP*.

Example

```
Lisp> (setf *print-right-margin* 60)
60
Lisp> (setf *print-miser-width* 35)
35
Lisp> (format t "~!Stars with Arabic names: ~:@!~S ~:_~S ~
                 ~27I~:_~S ~:I~@_~S ~_~S ~1I~_~S~.~."
          '(betelgeuse (deneb sirius vega)
            aldeberan algol (castor pollux) bellatrix))
Stars with Arabic names: BETELGEUSE
                         (DENEB SIRIUS VEGA)
                         ALDEBERAN
                         ALGOL
                         (CASTOR POLLUX)
                         BELLATRIX
```

- The text, “Stars with Arabic names:”, in the outer logical block causes the inner logical block to begin at column 26. With *PRINT-MISER-WIDTH_* set to 35, FORMAT enables miser mode when the logical block begins past column 25.
- FORMAT conserves space by starting a new line at every multiline mode new line directive (~_) and every if-needed new line directive (~:_).
- FORMAT starts a new line at the miser mode new line directive (~@_) and ignores the indentation directives (~nI).

PRINT-RIGHT-MARGIN Variable

Specifies the right margin for pretty-printing. Output may exceed this margin if you print long symbol names or strings, or if your FORMAT control string specifies no new line directives of any type. If the value of *PRINT-RIGHT-MARGIN* is NIL, the print function uses a value appropriate to the output device.

The WRITE and WRITE-TO-STRING functions have been extended in VAX LISP to accept the :RIGHT-MARGIN keyword. If you specify this keyword, *PRINT-RIGHT-MARGIN* is bound to the value you supply with the keyword before any output is produced.

See *VAX LISP Implementation and Extensions to Common LISP* for more information about using the *PRINT-RIGHT-MARGIN* variable.

Example

```
Lisp> (defun record-my-statistics
(name age siblings married?)
(unless (symbolp name)
(error "~S must be a symbol." name))
(setf (get name 'age) age
(get name 'number-of-siblings) siblings
(get name 'is-this-person-married?) married)
name)
RECORD-MY-STATISTICS
Lisp> (pprint-definition 'record-my-statistics)
(DEFUN RECORD-MY-STATISTICS (NAME AGE SIBLINGS MARRIED?)
  (UNLESS (SYMBOLP NAME) (ERROR "~S must be a symbol." NAME))
  (SETF (GET NAME' 'AGE) AGE
        (GET NAME 'NUMBER-OF-SIBLINGS) SIBLINGS
        (GET NAME 'IS-THIS-PERSON-MARRIED?) MARRIED)
  NAME)
Lisp> (setf *print-right-margin* 40)
40
Lisp> (pprint-definition 'record-my-statistics)
(DEFUN
 RECORD-MY-STATISTICS
 (NAME AGE SIBLINGS MARRIED?)
 (UNLESS
  (SYMBOLP NAME)
  (ERROR
   "~S must be a symbol."
   NAME))
 (SETF
  (GET NAME 'AGE) AGE
  (GET NAME 'NUMBER-OF-SIBLINGS)
  SIBLINGS
  (GET
   NAME
   'IS-THIS-PERSON-MARRIED?)
 MARRIED)
NAME)
```

- The call to the DEFUN macro defines a function named RECORD-MY-STATISTICS.
- The first call to the PPRINT-DEFINITION function shows the default output.
- The call to the SETF macro sets the value of the *PRINT-RIGHT-MARGIN* variable to 40.
- The second call to the PPRINT function shows what effect the variable's value has on the pretty-printed output. PPRINT-DEFINITION inserts new lines as needed before reaching column 40.

PRINT-SIGNALED-ERROR Function

Used by the VAX LISP error handler to display a formatted error message when an error is signaled. The function prints all output to the stream bound to the *ERROR-OUTPUT* variable. *VAX LISP Implementation and Extensions to Common LISP* describes the error message formats.

You can include a call to this function in an error handler that you create. See *VAX LISP Implementation and Extensions to Common LISP*.

Format

PRINT-SIGNALED-ERROR *function-name error-signaling-function*
&REST *args*

Arguments

function-name
The name of the function that is to call the specified error-signaling function.

error-signaling-function
The name of an error-signaling function. Valid function names are ERROR, CERROR, and WARN.

args
The specified error-signaling function's arguments.

Return Value

Unspecified.

PRINT-SIGNALED-ERROR Function

Example

```
Lisp> (defun continuing-error-handler (function-name
                                       error-signaling-function
                                       &rest args)
        (if (eq error-signaling-function 'cerror)
            (progn
              (apply #'print-signaled-error
                     function-name
                     error-signaling-function
                     args)
              (format *error-output*
                     "~&It will be continued automatically.~2%.")
              nil)
            (apply #'universal-error-handler
                   function-name
                   error-signaling-function
                   args)))
CONTINUING-ERROR-HANDLER
```

Defines an error handler that automatically continues from a continuable error after displaying an error message. All other errors are passed to the system's error handler.

PRINT-SLOT-NAMES-AS-KEYWORDS Variable

Determines how the slot names of a structure are formatted when they are displayed. The value can be T or NIL. If the value is T, slot names are preceded with a colon (:). For example:

```
#S(SPACE :AREA 4 :COUNT 10)
```

If the value is NIL, slot names are not preceded with a colon. For example:

```
#S(SPACE AREA 4 COUNT 10)
```

The default value is T.

Example

```
Lisp> (defstruct house
                 rooms
                 floors)
HOUSE
Lisp> (make-house :rooms 8 :floors 2)
#S(HOUSE :ROOMS 8 :FLOORS 2)
Lisp> (setf *print-slot-names-as-keywords* nil)
NIL
Lisp> (make-house :rooms 8 :floors 2)
#S(HOUSE ROOMS 8 FLOORS 2)
```

- The call to the DEFSTRUCT macro defines a structure named HOUSE.
- The first call to the constructor function MAKE-HOUSE creates a structure named HOUSE. Colons are included in the output, because the value of the *PRINT-SLOT-NAMES-AS-KEYWORDS* variable is T.

- The call to the SETF macro changes the value of the *PRINT-SLOT-NAMES-AS-KEYWORDS* variable to NIL.
- The second call to the constructor function MAKE-HOUSE creates a structure named HOUSE. Colons are *not* included in the output, because the value of the *PRINT-SLOT-NAMES-AS-KEYWORDS* variable is NIL.

REQUIRE Function

Examines the *MODULES* variable to determine if a specified module has been loaded. If the module is not loaded, the function loads the files that you specify for the module. If the module is loaded, its files are not reloaded.

When you call the REQUIRE function in VAX LISP, the function checks whether you explicitly specified pathnames that name the files it is to load. If you specify pathnames, the function loads the files the pathnames represent. If you do not specify pathnames, the function searches for the module's files in the following order:

1. The function searches the current directory for a source file or a fast-loading file with the specified module name. If the function finds such a file, it loads the file into the LISP environment. This search forces the function to locate a module you have created before the function locates a module of the same name that is present in one of the public places (see following steps).
2. If the logical name LISP$MODULES is defined, the function searches the directory to which this logical name refers for a source file or a fast-loading file with the specified module name. This search enables the VAX LISP sites to maintain a central directory of modules.
3. The function searches the directory to which the logical name LISP$SYSTEM refers for a source file or a fast-loading file with the specified module name. This search enables you to locate modules that are provided with the VAX LISP system.
4. If the function does not find a file with the specified module name, an error is signaled.

When you load a module, the pathname that refers to the directory that contains the module is bound to the *MODULE-DIRECTORY* variable. A description of the *MODULE-DIRECTORY* variable is provided earlier in this manual.

The REQUIRE function checks the *MODULES* variable to determine if a module has already been loaded. However, when loading a module, the REQUIRE function does not update the *MODULES* variable to indicate that the module has been loaded. The PROVIDE function (described in *Common LISP: The Language*) does update the *MODULES* variable. Use the PROVIDE function in a file containing a module to be loaded to indicate to the LISP system that the file contains a module of this name.

If the loaded file does not contain a corresponding PROVIDE, a subsequent REQUIRE of the module causes the file to be reloaded.

REQUIRE Function

Format

REQUIRE *module-name* **&OPTIONAL** *pathname*

Arguments

module-name
A string or a symbol that names the module whose files are to be loaded.

pathname
A pathname or a list of pathnames that represent the files to be loaded into LISP memory. The files are loaded in the same order the pathnames are listed, from left to right.

Return Value

Unspecified.

Example

```
Lisp> *modules*
("CALCULUS" "NEWTONIAN-MECHANICS")
Lisp> (require 'relative)
T
Lisp> *modules*
("RELATIVE" "CALCULUS" "NEWTONIAN-MECHANICS")
```

- The first evaluation of the `*MODULES*` variable shows that the modules CALCULUS and NEWTONIAN–MECHANICS are loaded.
- The call to the `REQUIRE` function checks whether the module RELATIVE is loaded. The previous evaluation of the `*MODULES*` variable indicated that the module was not loaded; therefore, the function loaded the module RELATIVE.
- The second evaluation of the `*MODULES*` variable shows that the module RELATIVE was loaded.

RIGHT-MARGIN Function

Returns the default right margin used by the pretty printer when printing to the stream. The current margin used by the pretty printer is controlled by the variable `*PRINT-RIGHT-MARGIN*`.

Format

RIGHT-MARGIN &OPTIONAL *output-stream*

Argument

output-stream

An output stream. The default value is *STANDARD-OUTPUT*. If you specify a value of T, the value of *TERMINAL-IO* is used.

Return Value

A non-negative fixnum indicating the default right margin for *output-stream*.

ROOM Function

Displays information about LISP memory. Information is displayed for the following memory spaces:

- Stack space
- Read-only space
- Static space
- Dynamic space
- Ephemeral space

For each space, the function provides the number of bytes (and segments) in use. For all spaces except stack, the function shows memory used by the data types listed in Table 11.

Table 11: ROOM Function Data Type Headings

Heading	Data Types
Cons	Conses
Boxed	Symbols, structures, arrays of element-type T
Unboxed	Strings, floats, bignums, arrays of element-type other than T, compiled code
Mixed	Functions, stacks

For the ephemeral areas, the ROOM function also shows the maximum number of segments that will be used for allocation. (See the description of AREA-SEGMENT-LIMIT.)

The following additional information is provided:

- The status of the full garbage collector, and the number of full collections since LISP image startup.
- The status of the ephemeral garbage collector, and the number of collections in each ephemeral area.
- The number of times LISP memory has been enlarged, and the number of segments that have been added.

Example

```
Lisp> (room-allocation)
1276831 ;
7602176
Lisp> (room-allocation :static)
2660502 ;
2818048
```

SET-TERMINAL-MODES Function

Sets the terminal characteristics of the stream bound to the *TERMINAL-IO* variable when you invoke the LISP system. Changes to the stream affect all streams attached to the terminal.

Be careful when you change the settings of terminal modes. A change to terminal modes affects all the streams that are open to the terminal. If you put a stream into pass-through mode, for example, all the streams open to the terminal are put into pass-through mode.

NOTE

> Create an error handler to prevent your terminal from being placed in a nonstandard state. See *VAX LISP Implementation and Extensions to Common LISP* for information about how to create an error handler.

Format

SET-TERMINAL-MODES &KEY :BROADCAST :ECHO :ESCAPE
:HALF-DUPLEX :PASS-ALL :PASS-THROUGH
:TYPE-AHEAD :WRAP

Arguments

:BROADCAST
Specifies whether the terminal can receive broadcast messages, such as MAIL notifications and REPLY messages. The value can be either T or NIL. If you specify T, the terminal can receive messages; if you specify NIL, the terminal cannot receive messages.

:ECHO
Specifies whether the terminal displays the input characters that it receives. The value can be either T or NIL. If you specify T, the terminal displays input characters; if you specify NIL, the terminal displays only data output from the system or from a user application program.

:ESCAPE

Specifies whether ANSI standard escape sequences that are transmitted from the terminal are handled as a single multicharacter terminator. The value can be either `T` or `NIL`. If you specify `T`, the escape sequences are handled as a single multicharacter terminator. The terminal driver checks the escape sequences for syntax before passing them to the program. For more information on escape sequences, see the *VMS I/O User's Reference Manual: Part I.*

:HALF-DUPLEX

Specifies the terminal's operating mode. The value can be either `T` or `NIL`. If you specify `T`, the terminal's operating mode is half-duplex. If you specify `NIL`, the operating mode is full-duplex. For a description of terminal operating modes, see the *VMS I/O User's Reference Manual: Part I.*

:PASS-ALL

Specifies whether the terminal is in pass-all mode. The value can be either `T` or `NIL`. If you specify `T`, the system does not expand tab characters to blanks, fill carriage return or line feed characters, recognize control characters, or receive broadcast messages. If you specify `NIL`, the system passes all data to an application program as binary data.

NOTE

> `:PASS-ALL` has been kept for compatibility with Version 1 of VAX LISP, but it is not recommended that you use `:PASS-ALL`.

:PASS-THROUGH

Specifies whether the terminal is in pass-through mode. The value can be either `T` or `NIL`. This mode is the same as the `:PASS-ALL` mode, except that "TTSYNC" protocol (Ctrl/S, Ctrl/Q) is still used.

:TYPE-AHEAD

Specifies whether the terminal accepts input that is typed when there is no outstanding read. The value can be either `T` or `NIL`. If you specify `T`, the terminal accepts input even if there is no outstanding read. If you specify `NIL`, the terminal is dedicated and accepts input only when a program or the system issues a read.

:WRAP

Specifies whether the terminal driver generates a carriage return and a line feed when the end of a line is reached. The value can be either `T` or `NIL`. If you specify `T`, the terminal driver generates a carriage return and a line feed when the end of a line is reached. The end of the line is determined by the terminal width setting.

Return Value

Unspecified.

Example

```
Lisp> (defvar *old-terminal-state*)
*OLD-TERMINAL-STATE*
Lisp> (defun pass-through-handler (function error &rest args)
        (let ((current-settings (get-terminal-modes)))
          (apply #'set-terminal-modes *old-terminal-state*)
          (apply #'universal-error-handler function error args)
          (apply #'set-terminal-modes current-settings)))
PASS-THROUGH-HANDLER
Lisp> (defun unusual-input nil
        (let ((*old-terminal-state* (get-terminal-modes))
              (*universal-error-handler* #'pass-through-handler))
          (unwind-protect (progn
                             (set-terminal-modes
                             :pass-through
                             t
                             :echo
                             nil)
                            (get-input))
                          (apply #'set-terminal-modes
                                 *old-terminal-state*))))
UNUSUAL-INPUT
```

- The call to the DEFVAR macro informs the LISP system that *OLD-TERMINAL-STATE* is a special variable.
- The first call to the DEFUN macro defines an error handler named PASS-THROUGH-HANDLER, which is used when the terminal is placed in an unusual state. The handler assumes that the normal terminal modes are stored as the value of the *OLD-TERMINAL-STATE* variable.
- The second call to the DEFUN macro defines a function named UNUSUAL-INPUT, which causes the function PASS-THROUGH-HANDLER to be the error handler while the function GET-INPUT is being executed. The GET-INPUT function is inside a call to the UNWIND-PROTECT function, so an error or throw puts the terminal back in its original state.

SHORT-SITE-NAME Function

Translates the logical name LISP$SHORT_SITE_NAME.

Format

SHORT-SITE-NAME

Argument

None.

Return Value

The translation of the logical name LISP$SHORT_SITE_NAME is returned as a string. If the logical name is not defined, `NIL` is returned.

Example

```
Lisp> (short-site-name)
"Smith's Computer Company"
```

SOFTWARE-VERSION-NUMBER Function

Returns as multiple values the version number of the specified software component.

Format

SOFTWARE-VERSION-NUMBER *component*

Argument

component
A string indicating the software component. Possible values are `"VAX LISP"`, `"VMS"`, and `"UIS"`. See the *VAX LISP/VMS DECwindows Programming Guide* for information on finding the version number of the X protocol.

Return Values

Multiple values. For a software version number in the form *x.y*:

1. A fixnum designating *x*
2. A fixnum designating *y*

Example

```
Lisp> (software-version-number "VAX LISP")
3 ;
0
Lisp>
```

SOURCE-CODE Function

Returns a lambda expression that is the source code for an interpreted function.

Format

SOURCE-CODE *function*

Argument

function
An interpreted function or a symbol designating an interpreted function.

Return Value

A lambda expression.

Example

```
Lisp> (defun f (x y)
        (* (+ x y) (* x y)))
F
Lisp> (pprint (symbol-function 'f))
#<Interpreted Function
   (LAMBDA (X Y) (BLOCK F (* (+ X Y) (* X Y))))
   4980172>
Lisp> (source-code (symbol-function 'f))
(LAMBDA (X Y) (BLOCK F (* (+ X Y) (* X Y))))
Lisp>
```

SPAWN Function

Creates a subprocess for executing Command Language Interpreter (CLI) commands. This function causes the LISP system to interrupt execution of a LISP process and optionally to execute the specified CLI command. If you specify the `:PARALLEL` keyword with a value of `T`, the LISP process continues to execute while the subprocess is executing. If you do not specify this keyword or if you specify it with `NIL`, the LISP process is put into a hibernation state until the subprocess completes its execution.

NOTE

> Be careful using this function with `:PARALLEL NIL` under DECwindows, both in the development environment and in your programs. Because this causes LISP to hibernate, no events can be processed. If events are queued and LISP does not respond within a server-defined timeout, the X server aborts the connection to the LISP process.

No such restriction applies to :PARALLEL T, because that does not make LISP hibernate.

This function is equivalent to the DCL SPAWN command. For more information on the SPAWN command, see the *VMS DCL Dictionary*.

Format

SPAWN &KEY :COMMAND-STRING :DCL-SYMBOLS :INPUT-FILE :LOGICAL-NAMES :OUTPUT-FILE :PARALLEL :PROCESS-NAME

Arguments

:COMMAND-STRING

A string specifying a DCL command that the specified subprocess is to process. The value must be a DCL command. By default, the SPAWN function does not process a command.

:DCL-SYMBOLS

Specifies whether the spawned subprocess is to acquire the currently defined CLI symbols from the LISP process. The value can be either T or NIL. If you specify T, the subprocess acquires the CLI symbols; if you specify NIL, the subprocess does not acquire the CLI symbols. The default value is T.

:INPUT-FILE

A pathname, namestring, symbol, or stream that specifies an input file containing one or more DCL commands to be associated with the logical name SYS$INPUT and to be executed by the spawned subprocess. If you specify both a command string and an input file, the command string is processed before the commands in the input file. The subprocess ends when processing is complete.

:LOGICAL-NAMES

Specifies whether the spawned subprocess is to acquire the currently defined logical names. The value can be either T or NIL. If you specify T, the subprocess acquires the logical names; if you specify NIL, the subprocess does not acquire the logical names. The default value is T.

:OUTPUT-FILE

Specifies a pathname, namestring, symbol, or stream that names the output file to be associated with the logical name SYS$OUTPUT and to which the results of the spawned subprocess are to be written.

:PARALLEL

Specifies whether the execution of the LISP system and the created subprocess are to be parallel. The value can be either T or NIL. If you specify T, the execution of the system and the subprocess are parallel; if you specify NIL, the LISP system remains in a hibernation state until the created subprocess completes its execution and exits. The default value is NIL.

:PROCESS-NAME

Specifies the name of the subprocess to be created. If you omit this keyword, the system generates a unique name.

Return Value

Unspecified.

Examples

1.
```
Lisp> (spawn)
$
```

   Creates a uniquely named subprocess and attaches the terminal to it. The commands typed at the terminal are directed to the subprocess until the subprocess exits.

2.
```
Lisp> (spawn :input-file "start.com"
             :output-file "start.log"
             :parallel t)
Lisp>
```

   Creates a subprocess that will execute the contents of START.COM.

3.
```
Lisp> (defun spawn-in-window
             (&optional (process-name nil))
        (let ((device-string
                  (uis:create-terminal
                          :banner-title
                          (or process-name "Subprocess"))))
          (spawn :input-file device-string
                 :output-file device-string
                 :process-name process-name
                 :parallel t)))
SPAWN-IN-WINDOW
Lisp> (spawn-in-window "Smith_1")
Lisp>
```

   This example works only on a VAXstation running UIS. It defines a function named SPAWN-IN-WINDOW that creates a process in a VAXstation terminal emulator window. The function UIS:CREATE-TERMINAL creates a terminal emulator window and returns the window's device name. By supplying this return value with the :INPUT-FILE and :OUTPUT-FILE keyword arguments, SPAWN-IN-WINDOW arranges for input to and output from the subprocess to be directed through the terminal emulator window. SPAWN-IN-WINDOW accepts an optional argument that becomes the name of the subprocess and the title of the window.

   When the SPAWN-IN-WINDOW function is called, a subprocess and a terminal emulator window named "Smith_1" are created. The cursor switches to the terminal emulator window. However, the user can switch the cursor back to the LISP prompt and continue to use LISP without logging out of the subprocess.

   See the *VAX LISP Interface to VWS Graphics* manual for information about the UIS:CREATE-TERMINAL function.

STEP Macro

Invokes the VAX LISP Stepper.

The STEP macro evaluates the form that is its argument and returns what the form returns. In the process, you can interactively step through the evaluation of the form. Entering a question mark (?) in response to the Stepper prompt displays helpful information. The Stepper is command oriented rather than expression oriented—do not enclose commands within parentheses.

For further information on using the VAX LISP Stepper, see Chapter 4 of the *VAX LISP/VMS Program Development Guide*.

Format

STEP *form*

Argument

form

A form to be evaluated.

Return Value

The value returned by the *form* argument.

Example

```
Lisp> (step (factorial 3))
: #9: (FACTORIAL 3)
Step >
```

Invokes the VAX LISP Stepper for the function call (FACTORIAL 3).

STEP-ENVIRONMENT Variable

The *STEP-ENVIRONMENT* variable, a debugging tool, is bound to the lexical environment in which *STEP-FORM* is being evaluated. By default in the Stepper, the lexical environment is used if you use the EVALUATE command. See *Common LISP: The Language* for a description of dynamic and lexical environment variables.

Some Common LISP functions (for example, EVALHOOK, APPLYHOOK, and MACROEXPAND) take an optional environment argument. The value bound to the *STEP-ENVIRONMENT* variable can be passed as an environment to these functions to allow evaluation of forms in the context of the stepped form.

Example

```
Step> eval *step-form*
(FIBONACCI (- X 1))
Step> (evalhook 'x nil nil nil)
"Top level value of X"
Step> (evalhook 'x nil nil *step-environment*)
3
```

The *STEP-ENVIRONMENT* variable in this call to the EVALHOOK function causes the local value of X to be used in the evaluation of the form (- X 1). See Chapter 4 of the *VAX LISP/VMS Program Development Guide* for the full Stepper sessions from which this excerpt is taken.

STEP-FORM Variable

The *STEP-FORM* variable, a debugging tool, is bound to the form being evaluated by the VAX LISP Stepper. For example, while executing the form:

```
(STEP (FUNCTION-Z ARG1 ARG2))
```

the value of *STEP-FORM* is the list (FUNCTION-Z ARG1 ARG2). When not stepping, the value is undefined.

Example

```
Step> step
: : : : : : : : #39: X => 4
: : : : : : : #35: => NIL
: : : : : : : #34: (+ FIBONACCI (- X 1)) (FIBONACCI (- X 2))
Step> step
: : : : : : : : #38: (FIBONACCI (- X 1))
Step> eval *step-form*
(FIBONACCI (- X 1))
```

See Chapter 4 of the *VAX LISP/VMS Program Development Guide* for the full Stepper session from which this excerpt is taken. In this case, the *STEP-FORM* variable is bound to (FIBONACCI (- X 1)).

SUSPEND Function

Writes information about a LISP system to a file, making it possible to resume the LISP system at a later time. The function does not stop the current system, but copies the state of the LISP system when the function is invoked to the specified file. When you reinvoke the LISP system with the /RESUME qualifier and the file name that was specified with the SUSPEND function, program execution continues from the point where the SUSPEND function was called.

Only the static and dynamic portions of the LISP environment are written to the specified file. When you resume a suspended system, the read-only sections of the LISP environment are taken from LISP$SYSTEM:LISP.EXE. You must make sure that your original LISP system is in LISP$SYSTEM:LISP.EXE; if it is not, you will be unable to resume the system.

When a suspended system is resumed, the LISP environment is identical to the environment that existed when the suspend operation occurred, with the following exceptions:

- All streams and window streams are closed except the standard streams (`*STANDARD-INPUT*`, `*STANDARD-OUTPUT*`, and so on).
- The `*DEFAULT-PATHNAME-DEFAULTS*` variable is set to the current directory.
- Callout state might be lost. (See Chapter 4 of the *VAX LISP/VMS System Access Guide.*)
- Any interrupt functions are uninstated. (See Chapter 6 of the *VAX LISP/VMS System Access Guide.*) They are not automatically reinstated upon resuming.
- All state associated with the DECwindows-based development environment's utilities is lost. On resuming the Listener will be reinitialized and brought up using the last saved defaults. (See Chapter 7 of the *VAX LISP/VMS Program Development Guide.*)
- For all workstation-related functions that take an *action* argument, the *action* is reset to the system default state. An *action* that you have established is not automatically reestablished upon resuming.
- Some Editor state is changed. (See the *VAX LISP/VMS Editor Programming Guide.*)
- In a LISP with user-programmed CLX or XUI toolkit connections, all non-LISP state is lost. This includes displays, windows, and widgets. (See the *VAX LISP/VMS DECwindows Programming Guide.*)
- On a VWS/UIS workstation, windows, displays, and display lists are lost.

Suspended systems must be resumed from the same LISP system that suspended them. For LISP systems created with the System-Building Utility, the LISP system that resumes a suspended system must be the same image as the LISP system that suspended the system or a copy of that image. Two LISP systems created at different times cannot resume systems suspended by the other, even if the two LISP systems were created with identical calls to `DEFINE-LISP-SYSTEM`.

The `SUSPEND` function should not be called during a call to `LOAD`; the stream used by `LOAD` cannot be re-created to finish the load on resuming the LISP.

Format

SUSPEND *pathname*

Argument

pathname

A pathname, namestring, or symbol that represents the file name to which the function writes the LISP-system state.

Return Value

T, when the LISP system is resumed at a later time; NIL, when execution continues after a suspend operation.

Example

```
Lisp> (defun program-main-loop nil
        (loop (princ "Enter number> ")
              (setf x (read *standard-input*))
              (format *standard-output*
                      "~%The square root of ~F is ~F. ~%"
                      x
                      (sqrt x))))
PROGRAM-MAIN-LOOP
Lisp> (defun dump-program nil
        (suspend "myprog.sus")
        (fresh-line)
        (princ "Welcome to my program!")
        (terpri)
        (program-main-loop))
DUMP-PROGRAM
Lisp> (dump-program)
; Starting full GC ...
; ... Full GC finished
Welcome to my program!
Enter number> 25
The square root of 25.0 is 5.0.
Enter number> 5
The square root of 5.0 is 2.236038.
Enter number>
   .
   .
   .
CTRL/C
Lisp> (exit)
$ lisp/resume=myprog.sus
Welcome to my program!
Enter number>
```

- The first call to the DEFUN macro defines a function named PROGRAM-MAIN-LOOP.
- The second call to the DEFUN macro defines a function named DUMP-PROGRAM.
- The call to the DUMP-PROGRAM function copies the current state of the LISP environment to the file MYPROG.SUS. The LISP system continues to run, displaying the message "Welcome to my program!" and then executes the PROGRAM-MAIN-LOOP function.
- The call to the EXIT function exits the LISP system.
- The LISP/RESUME=MYPROG.SUS specification reinvokes the LISP system, displays the message, and executes the PROGRAM-MAIN-LOOP function.

TIME Macro

Evaluates a form, displays the form's CPU time and real time, and returns the values that the form returns.

The time information is displayed in the following format:

CPU Time: *cpu-time* sec., Real Time: *real-time* sec.

If garbage collections occur during the evaluation of a call to the `TIME` macro, the macro displays another line of time information. This line includes information about the CPU time and real time used by the garbage collector.

Format

TIME *form*

Argument

form
The form that is to be evaluated.

Return Value

The form's return values are returned.

Example

```
Lisp> (time (test))
CPU Time: 0.03 sec., Real Time: 0.23 sec.
6
```

TOP-LEVEL-PROMPT Variable

Lets you change the top-level prompt. The value of this variable can be:

- A string
- A function of no arguments that returns a string
- `NIL`

If you specify `NIL`, the default prompt, Lisp>, is used.

Example

```
Lisp> (setf *top-level-prompt*  "TOP> ")
"TOP> "
TOP>
```

Sets the value of the variable *TOP-LEVEL-PROMPT* to "TOP> ".

TRACE Macro

Enables tracing for one or more functions and macros.

VAX LISP allows you to specify a number of options that suppress the TRACE macro's displayed output or that cause additional information to be displayed. The options are specified as keyword-value pairs. The keyword-value pairs you can specify are listed in Table 12.

NOTE

> The arguments specified in a call to the TRACE macro are not evaluated when the call to TRACE is executed. Some forms are evaluated repeatedly, as described below.

Format

TRACE &REST *trace-description*

Argument

trace-description
Zero or more optional arguments. If no argument is specified, the TRACE macro returns a list of functions and macros that are currently being traced. Trace-description arguments can be specified in three formats:

- One or more function and/or macro names can be specified, which enables tracing for that function(s) and/or macro(s).

  name-1 name-2 . . .

- The name of each function or macro can be specified with keyword-value pairs. The keyword-value pairs specify the operations that the TRACE macro is to perform when it traces the specified function or macro. The name and the keyword-value pairs must be specified as a list whose first element is the function or macro name.

  (*name keyword-1 value-1 keyword-2 value-2 . . .*)

- A list of function and/or macro names can be specified with keyword-value pairs. The keyword-value pairs specify the operations that the TRACE macro is to perform when it traces each function and/or macro in the list. The list of names and the keyword-value pairs must be specified as a list whose first element is the list of names.

((name-1 name-2 . . .) keyword-1 value-1
keyword-2 value-2 . . .)

Table 12 lists the keywords and values that can be specified. The forms that are referred to in the value descriptions are evaluated in the null lexical environment and the current dynamic environment.

Table 12: TRACE Options

Keyword-Value Pair	Description
`:DEBUG-IF` *form*	Specifies a form that is to be evaluated before and after each call to the specified function or macro. If the form returns a value other than `NIL`, the VAX LISP Debugger is invoked before and after the function or macro is called.
`:PRE-DEBUG-IF` *form*	Specifies a form that is to be evaluated before each call to the specified function or macro. If the form returns a value other than `NIL`, the VAX LISP Debugger is invoked before the specified function or macro is called.
`:POST-DEBUG-IF` *form*	Specifies a form that is to be evaluated after each call to the specified function or macro. If the form returns a value other than `NIL`, the VAX LISP Debugger is invoked after the specified function or macro is called.
`:PRINT` *form-list*	Specifies a list of forms that are to be evaluated and whose values are to be displayed before and after each call to the specified function or macro. The values are displayed one per line and are indented to match other output displayed by the `TRACE` macro.
`:PRE-PRINT` *form-list*	Specifies a list of forms that are to be evaluated and whose values are to be displayed before each call to the specified function or macro. The values are displayed one per line and are indented to match other output displayed by the `TRACE` macro.
`:POST-PRINT` *form-list*	Specifies a list of forms that are to be evaluated and whose values are to be displayed after each call to the specified function or macro. The values are displayed one per line and are indented to match other output displayed by the `TRACE` macro.
`:STEP-IF` *form*	Specifies a form that is to be evaluated before each call to the specified function or macro. If the form returns a value other than `NIL`, the VAX LISP Stepper is invoked and the function or macro is stepped through.
`:SUPPRESS-IF` *form*	Specifies a form that is to be evaluated before each call to the specified function or macro. If the form returns a value other than `NIL`, the `TRACE` macro does not display the arguments and the return value of the specified function or macro.

(continued on next page)

Table 12 (Cont.): TRACE Options

Keyword-Value Pair	Description
`:DURING` *name*	Specifies a function or macro name or a list of function and macro names. The function or macro specified by the `TRACE` function is traced only when it is called (directly or indirectly) from within one of the functions or macros specified by the `:DURING` keyword.

See the *VAX LISP/VMS Program Development Guide* for more information on the VAX LISP Debugger, Stepper, and Tracer.

Return Value

A list of the functions currently being traced.

Examples

1.
```
Lisp> (trace factorial count1 count2)
(FACTORIAL COUNT1 COUNT2)
```

Enables the VAX LISP Tracer for the functions `FACTORIAL`, `COUNT1`, and `COUNT2`.

2.
```
Lisp> (trace)
(FACTORIAL COUNT1 COUNT2)
```

Returns a list of the functions for which the Tracer is enabled.

3.
```
Lisp> (defun reverse-count (n)
        (declare (special *go-into-debugger*))
        (if (> n 3)
            (setq *go-into-debugger* t)
            (setq *go-into-debugger* nil))
        (cond ((= n 0) 0)
              (t (print n) (+ 1 (reverse-count (- n 1))))))
REVERSE-COUNT
Lisp> (setq *go-into-debugger* nil)
NIL
Lisp> (reverse-count 3)
3
2
1
3
Lisp> (trace (reverse-count :debug-if *go-into-debugger*))
(REVERSE-COUNT)
Lisp> (reverse-count 3)
#4: (REVERSE-COUNT 3)
3
. #16: (REVERSE-COUNT 2)
2
. . #28: (REVERSE-COUNT 1)
1
. . . #40: (REVERSE-COUNT 0)
. . . #40=> 0
. . #28=> 1
. #16=> 2
```

```
#4=> 3
3
Lisp> (reverse-count 4)
#4:
4
. #16: (REVERSE-COUNT 3)
Control Stack Debugger
Apply #17: (DEBUG)
Debug 1> continue
3
. . #28: (REVERSE-COUNT 2)
2
. . . #40: (REVERSE-COUNT 1)
1
. . . . #52: (REVERSE-COUNT 0)
. . . . #52=> 0
. . . #40=> 1
. . #28=> 2
. #16=> 3
#4=> 4
4
Lisp>
```

The recursive function REVERSE-COUNT is defined to count down from the number it is given and to return that number after the function is evaluated. However, if the given number is greater than 3 (set low to simplify the example), the global variable *GO-INTO-DEBUGGER* (preset to NIL) is set to T.

The first time the REVERSE-COUNT function is traced by use of the :DEBUG-IF keyword, the argument is 3. The second time the function is traced, the argument is over 3. This sets the global variable *GO-INTO-DEBUGGER* to T, which causes the Debugger to be invoked during a trace of the REVERSE-COUNT function. The Debugger is invoked after the function's argument is evaluated.

To reset the global variable *GO-INTO-DEBUGGER* to NIL, the REVERSE-COUNT function must be completed. So, the evaluation of the function was continued with the Debugger CONTINUE command.

4.

```
Lisp> (trace (reverse-count
                 :pre-debug-if *go-into-debugger*))
(REVERSE-COUNT)
Lisp> (reverse-count 4)
#4: (REVERSE-COUNT 4)
4
. #16: (REVERSE-COUNT 3)
Control Stack Debugger
Apply #17:
Debug 1>
```

The 4 argument to the REVERSE-COUNT function causes the *GO-INTO-DEBUGGER* variable to be set to T, which in turn causes the Debugger to be invoked before the first recursive call to the REVERSE-COUNT function.

5.

```
Lisp> (trace (reverse-count
               :post-debug-if *go-into-debugger*))
(REVERSE-COUNT)
Lisp> (reverse-count 4)
#4: (REVERSE-COUNT 4)
4
. #16: (REVERSE-COUNT 3)
3
. . #28: (REVERSE-COUNT 2)
2
. . . #40: (REVERSE-COUNT 1)
1
. . . . #52: (REVERSE-COUNT 0)
. . . . #52=> 0
. . . #40=> 1
. . #28=> 2
. #16=> 3
#4=> 4
4
Lisp> (trace (reverse-count
               :post-debug-if (not *go-into-debugger*)))
(REVERSE-COUNT)
Lisp> (reverse-count 4)
#4: (REVERSE-COUNT 4)
4
. #16: (REVERSE-COUNT 3)
3
. . #28: (REVERSE-COUNT 2)
2
. . . #40: (REVERSE-COUNT 1)
1
. . . . #52: (REVERSE-COUNT 0)
Control Stack Debugger
Apply #53: (DEBUG)
Debug 1> continue
. . . . #52=> 0
Control Stack Debugger
Apply #41: (DEBUG)
Debug 1> continue
. . . #40=> 1
Control Stack Debugger
Apply #29: (DEBUG)
Debug 1> continue
. . #28=> 2
Control Stack Debugger
Apply #17: (DEBUG)
Debug 1> continue
. #16=> 3
Control Stack Debugger
Apply #5: (DEBUG)
Debug 1> continue
#4=> 4
4
Lisp>
```

Here, the first time the REVERSE-COUNT function is evaluated, the Debugger is not invoked despite the :POST-DEBUG-IF keyword, because the keyword invokes the Debugger only if its condition is met after the function is evaluated. However, after the function is evaluated, the *GO-INTO-DEBUGGER* variable is reset to NIL. If the form (setq *go-into-debugger* nil) were removed from the definition of the REVERSE-COUNT function, the variable would not have been reset to NIL, and the Debugger would have been invoked.

The second time the REVERSE-COUNT function is invoked, the form (not *go-into-debugger*) evaluates to T, since the value of its argument is NIL. This gives the :POST-DEBUG-IF keyword a T value, which in turn fulfills the condition of invoking the Debugger after the function is evaluated.

In this situation, the Debugger CONTINUE command causes only one evaluation. Here, the CONTINUE command must be repeated to evaluate all the recursive calls. This example differs from Example 1, where the CONTINUE command did not have to be repeated.

6.
```
Lisp> (setf *L* 5 *M* 6 *N* 7)
7
Lisp> (trace (* :print (*L* *M* *N*)))
(*)
Lisp> (+ 2 3 *L* *M* *N*)
23
Lisp> (* 2 3 *L* *M* *N*)
#4: (* 2 3 5 6 7)
#4  *L* is 5
#4  *M* is 6
#4  *N* is 7
#4=> 1260
#4  *L* is 5
#4  *M* is 6
#4  *N* is 7
1260
```

The + function is not traced, but the * function is traced. The values of the global variables *L*, *M*, and *N* are displayed before and after the call to the * function is evaluated.

7.
```
Lisp> (trace (* :pre-print (*L* *M* *N*)))
(*)
Lisp> (* 2 3 *L* *M* *N*)
#4: (* 2 3 5 6 7)
#4  *L* is 5
#4  *M* is 6
#4  *N* is 7
#4=> 1260
1260
```

The values of the global variables *L*, *M*, and *N* are displayed before the call to the * function is evaluated.

8.
```
Lisp> (trace (* :post-print (*L* *M* *N*)))
(*)
Lisp> (* 2 3 *L* *M* *N*)
#4: (* 2 3 5 6 7)
#4=> 1260
#4  *L* is 5
#4  *M* is 6
#4  *N* is 7
1260
```

The values of the global variables *L*, *M*, and *N* are displayed after the call to the * function is evaluated.

9.
```
Lisp> (trace +)
(+)
Lisp> (+ 2 3 (square 4) (sqrt 25))
#4: (+ 2 3 16 5.0)
#4=> 26.0
26.0
Lisp> (setq *stop-tracing* t)
T
Lisp> (trace (+ :suppress-if *stop-tracing*))
(+)
Lisp> (+ 2 3 (square 4) (sqrt 25))
26.0
```

The first call to the + function is traced. The second call to the + function is not traced because of the form `(+ :suppress-if *stop-tracing*)`.

10.
```
Lisp> (trace (factorial :step-if t))
(FACTORIAL)
Lisp> (+ (factorial 2) 3)
#6: (FACTORIAL 2)
#10: (BLOCK FACTORIAL (IF (<= N 1) 1
                          (* N (FACTORIAL (- N 1)))))
Step>
: #15: (IF (<= N 1) 1 (* N (FACTORIAL (- N 1))))
Step>
: : #20: (<= N 1)
Step>
    .
    .
    .
```

The call to the `FACTORIAL` function invokes the Stepper.

11.
```
Lisp> (trace (list-length :during print-length))
(LIST-LENGTH)
Lisp> (print-length '(cat dog pony))
#13: (LIST-LENGTH (CAT DOG PONY))
#13=> 3

The length of (CAT DOG PONY) is 3.
NIL
```

The `PRINT-LENGTH` function has been defined to find the length of its argument with the function `LIST-LENGTH`. The `LIST-LENGTH` function is traced during the call to the `PRINT-LENGTH` function.

12.
```
Lisp> (defun fibonacci (x)
        (if (< x 3) 1
            (+ (fibonacci (- x 1)) (fibonacci (- x 2)))))
FIBONACCI
Lisp> (trace (fibonacci
                :pre-debug-if (< (second *trace-call*) 2)
                :suppress-if t))
(FIBONACCI)
Lisp> (fibonacci 5)
Control Stack Debugger
Apply #30: (DEBUG)
Debug 1> down
Eval  #27: (FIBONACCI (- X 2))
Debug 1> down
Eval  #26: (+ (FIBONACCI (- X 1))
              (FIBONACCI (- X 2)))
Debug 1> down
Eval  #25: (IF (< X 3) 1
               (+ (FIBONACCI (- X 1))
                  (FIBONACCI (- X 2))))
Debug 1> down
Eval  #24: (BLOCK FIBONACCI
               (IF (< X 3) 1
                   (+ (FIBONACCI (- X 1))
                      (FIBONACCI (- X 2)))))
Debug 1> down
Apply #22: (FIBONACCI 3)
Debug 1> (cadr (debug-call))
3
Debug 1> continue
Control Stack Debugger
Apply #22:
Debug 1> continue
5
```

This example illustrates the following points:

- First, FIBONACCI is defined.
- Then the TRACE macro is called for FIBONACCI. TRACE is specified to invoke the Debugger if the first argument to FIBONACCI (the second element of *TRACE-CALL*) is less than 2. Since the :PRE-DEBUG-IF option is specified, the Debugger is invoked before the call to FIBONACCI. As the :SUPPRESS-IF option has a value of T, calls to FIBONACCI cause no trace output.
- The DOWN commands move the pointer down the control stack.
- The DEBUG-CALL function returns a list representing the current debug frame function call. In this case, the CADR of the list is 3. This accesses the first argument to the function in the current stack frame.
- Finally, the CONTINUE command continues the evaluation of FIBONACCI.

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

TRACE is called for FIBONACCI to start the Debugger if the returned value (which is bound to *TRACE-VALUES*) exceeds 2. The returned value exceeds 2 twice—once when it returns 3 and at the end when it returns 5.

TRACE-CALL Variable

The *TRACE-CALL* variable, a debugging tool, is bound to the function or macro call being traced.

Examples

These examples assume that the function FIBONACCI has already been defined. See the examples of the TRACE macro for the definition.

1.
```
Lisp> (trace (fibonacci
               :suppress-if (> (second *trace-call*) 1)))
(FIBONACCI)
```

This causes FIBONACCI to be traced only if its first argument is 1 or less.

2.
```
Lisp> (trace (fibonacci
               :suppress-if (<= (length *trace-call*) 2)))
(FIBONACCI)
```

This causes FIBONACCI to be traced if it is called with more than one argument.

3.
```
Lisp> (trace (fibonacci
               :predebug-if (< (second *trace-call*) 2)
               :suppress-if (< (second *trace-call*) 2)))
(FIBONACCI)
```

The TRACE macro is enabled for FIBONACCI. In this case, the Debugger is invoked and tracing suppressed if the first argument to FIBONACCI (the SECOND of the value of the *TRACE-CALL* variable) is less than 2. So, for example, if FIBONACCI is called with the arguments 3 and 5, *TRACE-CALL* is bound to the form (fibonacci 3 5); as 3 is greater than 2, the call is traced and the Debugger not entered. See the description of the TRACE macro for further examples of the use of *TRACE-CALL*.

TRACE-VALUES Variable

The *TRACE-VALUES* variable, a debugging tool, is bound to the list of values returned by the traced function. You can use the value bound to this variable in the forms used with the trace option keywords, such as :DEBUG-IF.

Example

This example assumes that the FACTORIAL function has already been defined.

```
Lisp> (trace (factorial :post-print *trace-values*))
Lisp> (FACTORIAL)
Lisp> (factorial 4)
#5: (FACTORIAL 4)
. #14: (FACTORIAL 3)
. . #23: (FACTORIAL 2)
. . . #32: (FACTORIAL 1)
. . . #32=> 1
. . . #32  *TRACE-VALUES* is (1)
. . #23=> 2
. . #23  *TRACE-VALUES* is (2)
. #14=> 6
. #14  *TRACE-VALUES* is (6)
#5=> 24
#5  *TRACE-VALUES* is (24)
24
```

The values returned by the FACTORIAL function are bound to the *TRACE-VALUES* variable and are displayed as (1), (2), (6), and (24). Since the *TRACE-VALUES* variable is bound to the list of values returned by a function, it can be used only in the :POST- options to the TRACE macro. Before being bound to the return values, it returns NIL. See the description of the TRACE macro for further examples of the use of the *TRACE-VALUES* variable.

TRANSLATE-LOGICAL-NAME Function

Searches a logical name table for a logical name, translates it, and returns it as a list of strings.

The TRANSLATE-LOGICAL-NAME function performs only one level of logical name translation.

This function is equivalent to the DCL SHOW LOGICAL command. For additional information about the SHOW LOGICAL command or about using logical names, see the *VMS DCL Dictionary*.

Format

TRANSLATE-LOGICAL-NAME *string*
&KEY :TABLE :CASE-SENSITIVE

Arguments

string
The logical name for which the function is to search.

:TABLE
The logical name table that the function is to search. If you do not specify a table name, the process, group, system, and VMS DECwindows name tables are searched in that order. The values you can specify with the :TABLE keyword are the following:

:PROCESS	Process name table (LNM$PROCESS_TABLE).
:GROUP	Group name table (LNM$GROUP).
:SYSTEM	System name table (LNM$SYSTEM_TABLE).
:DECWINDOWS	DECwindows name table (DECW$LOGICAL_NAMES).
:ALL	Search all four tables (LNM$DCL_LOGICAL). This is the default.

You can also specify a string that names a table created with the DCL command CREATE/NAME_TABLE.

:CASE-SENSITIVE

Used to restrict the search to a case-sensitive search. Valid values are T for case-sensitive search or NIL for case-insensitive search. The default is NIL. Use a value of T if you have multiple logical names that differ only in case.

Return Value

If the logical name has any translations, they are returned as a list of strings. If no match is found, NIL is returned.

Example

```
Lisp> (defun show-where-i-am (&optional
                               (stream *standard-output*))
        (format stream
                "~&Current host is ~A ~
                 ~%Current device is ~A ~
                 ~%Current directory is ~A ~%"
                (car (translate-logical-name "sys$node"))
                (car (translate-logical-name "sys$disk"))
                (concatenate 'string
                             "["
                             (pathname-directory
                              (default-directory))
                             "]"))
        (values))
SHOW-WHERE-I-AM
Lisp> (show-where-i-am)
Current host is MIAMI::
Current device is DBA1:
Current directory is [VAXLISP]
Lisp> (setf (default-directory) "SYS$LIBRARY")
"SYS$LIBRARY"
Lisp> (show-where-i-am)
Current host is MIAMI::
Current device is SYS$SYSROOT:
Current directory is [SYSLIB]
```

- The call to the DEFUN macro defines a function named SHOW-WHERE-I-AM, which displays the current host, device, and directory.
- The first call to the function SHOW-WHERE-I-AM displays the current host, device, and directory.

- The call to the SETF macro changes the directory to SYSLIB.
- The second call to the function SHOW-WHERE-I-AM includes the new directory in the output that the function displays.

UNBIND-KEYBOARD-FUNCTION Function

Removes the binding of a function from a control character.

Format

UNBIND-KEYBOARD-FUNCTION *control-character*

Argument

control-character
The control character from which a function's binding is to be removed.

Return Value

T, if a binding is removed; NIL, if the control character is not bound to a function.

Example

```
Lisp> (bind-keyboard-function #\^B #'break)
T
Lisp> (unbind-keyboard-function #\^B)
T
```

- The call to the BIND-KEYBOARD-FUNCTION function binds Ctrl/B to the BREAK function.
- The call to the UNBIND-KEYBOARD-FUNCTION function removes the binding of the function that is bound to Ctrl/B.

UNCOMPILE Function

Restores the interpreted function definition of a symbol, if the symbol's definition was compiled with a call to the COMPILE function.

The UNCOMPILE function is useful for editing function definitions and debugging. For example, if you are dissatisfied with the results of a function compilation, you can uncompile the function, edit it, and then recompile it.

UNCOMPILE Function

NOTE

You cannot uncompile:

- System functions and macros
- Functions and macros that were loaded from files compiled by the COMPILE-FILE function
- Functions and macros that were loaded from files compiled by the DCL /COMPILE qualifier of the LISP command

Format

UNCOMPILE *symbol*

Argument

symbol
The symbol that represents the function that is to be uncompiled.

Return Value

The name of the restored function, if the specifed symbol represents an existing compiled lambda expression and has an interpreted definition; NIL, if it does not.

Example

```
Lisp> (defun add2 (first second) (+ first second))
ADD2
Lisp> (compile 'add2)
ADD2 compiled.
ADD2
Lisp> (compiled-function-p #'add2)
T
Lisp> (uncompile 'add2)
ADD2
Lisp> (compiled-function-p #'add2)
NIL
```

- The call to the DEFUN macro defines the function ADD2.
- The call to the COMPILE function compiles the function ADD2.
- The call to the UNCOMPILE function successfully restores the interpreted definition of the function ADD2, because the function is defined and was compiled with the COMPILE function.

UNDEFINE-LIST-PRINT-FUNCTION Macro

Disables the list-print function defined for a symbol. If another list-print function was superseded by the undefined list-print function, the older function is reenabled; otherwise, no other list-print function exists for the given symbol.

See *VAX LISP Implementation and Extensions to Common LISP* for more information about list-print functions.

Format

UNDEFINE-LIST-PRINT-FUNCTION *symbol*

Argument

symbol
The name of the list-print function that is to be disabled.

Return Value

The name of the disabled list-print function.

Example

This example assumes that a list-print function MY-SETQ has already been defined with the DEFINE-LIST-PRINT-FUNCTION macro.

```
Lisp> (undefine-list-print-function my-setq)
MY-SETQ
```

Undefines the list-print function named MY-SETQ.

UNINSTATE-INTERRUPT-FUNCTION Function

Informs LISP that the interrupt function identified by *iif-id* will no longer be used. The *iif-id* can no longer be given as the *astprm* argument to a routine that can cause an AST. However, UNINSTATE-INTERRUPT-FUNCTION does not prevent a routine from causing an AST with that *iif-id*. For example, an external routine that was called with that *iif-id* before the use of UNINSTATE-INTERRUPT-FUNCTION might later cause an AST. VAX LISP ignores ASTs for which the corresponding interrupt function has been uninstated.

Format

UNINSTATE-INTERRUPT-FUNCTION *iif-id*

Argument

iif-id

An interrupt function identifier previously returned by `INSTATE-INTERRUPT-FUNCTION`.

Return Value

Unspecified.

Examples

1.
```
Lisp> (uninstate-interrupt-function timer-iif)
T
```

Makes the interrupt function represented by `timer-iif` unavailable for future use.

2.
```
      .
      .
      .
(let ((button-iif (instate-interrupt-function
                    #'button-handler)))
  (uis:set-button-action display window button-iif)
      .
      .
      .
  (uis:set-button-action display window nil)
  (uninstate-interrupt-function button-iif)))
```

In this code fragment, the interrupt function `BUTTON-HANDLER` is instated and `BUTTON-IIF` is bound to its *iif-id*. The first call to `SET-BUTTON-ACTION` establishes `BUTTON-HANDLER` as the function to execute when a workstation pointer button is pressed or released. Later, the second call to `SET-BUTTON-ACTION` requests that no action be taken when buttons are pressed or released. Finally, `UNINSTATE-INTERRUPT-FUNCTION` removes the interrupt function represented by `BUTTON-IIF` from the system.

UNIVERSAL-ERROR-HANDLER Function

By default, this function handles all errors that are signaled. The VAX LISP `*UNIVERSAL-ERROR-HANDLER*` variable is initially bound to this function.

VAX LISP Implementation and Extensions to Common LISP describes the VAX LISP error handler.

Format

UNIVERSAL-ERROR-HANDLER *function-name error-signaling-function*
&REST *args*

UNIVERSAL-ERROR-HANDLER Function

Arguments

function-name
The name of the function that produced or signaled the error.

error-signaling-function
The name of an error-signaling function. Valid function names are ERROR, CERROR, and WARN.

args
The arguments of *error-signaling-function*.

Return Value

Invokes the VAX LISP Debugger, exits the LISP system, or returns NIL.

Example

```
Lisp>  (defun critical-error-handler (function-name
                                      error-signaling-function
                                      &rest args)
         (when (or (eq error-signaling-function 'error)
                   (eq error-signaling-function 'cerror))
               (flash-alarm-light))
         (apply #'universal-error-handler
                function-name
                error-signaling-function
                args))
CRITICAL-ERROR-HANDLER
```

Defines an error handler that checks whether a fatal or continuable error is signaled. If either type of error is signaled, the handler flashes an alarm light and then passes the error signal information to the universal error handler.

For more information on how to create an error handler, see *VAX LISP Implementation and Extensions to Common LISP*.

UNIVERSAL-ERROR-HANDLER Variable

By default, this variable is bound to the VAX LISP error handler, the UNIVERSAL-ERROR-HANDLER function. If you create an error handler, you must bind the *UNIVERSAL-ERROR-HANDLER* variable to it.

Example

```
Lisp>  (defun critical-error-handler (function-name
                                      error-signaling-function
                                      &rest args)
         (when (or (eq error-signaling-function 'error)
                   (eq error-signaling-function 'cerror))
               (flash-alarm-light))
         (apply #'universal-error-handler
                function-name
                error-signaling-function
                args))
CRITICAL-ERROR-HANDLER
Lisp>  (let ((*universal-error-handler*
              #'critical-error-handler))
         (perform-critical-operation))
```

- The call to the DEFUN macro defines an error handler named CRITICAL-ERROR-HANDLER.
- The call to the special form LET binds the *UNIVERSAL-ERROR-HANDLER* variable to the error handler named CRITICAL-ERROR-HANDLER, while the PERFORM-CRITICAL-OPERATION function is evaluated.

VMS-DEBUG Function

Invokes the VMS Symbolic Debugger. Use this function to invoke the Symbolic Debugger or to activate a shareable image and use the Debugger before calling out to an external routine that you want to debug. See the *VAX LISP/VMS Program Development Guide* for an example of using this function.

Format

VMS-DEBUG &KEY :EXTERNAL-ROUTINE :COMMAND-LINE

Arguments

:EXTERNAL-ROUTINE
A symbol naming the external routine you wish to debug. When you supply this argument, the image containing the external routine is activated, and the Symbolic Debugger executes the SET IMAGE command on the image.

:COMMAND-LINE
A string containing one or more Symbolic Debugger commands. Separate commands with semicolons. If you have supplied a value for the :EXTERNAL-ROUTINE argument, the resulting SET IMAGE command executes before the commands you supply with the :COMMAND-LINE argument.

Return Value

Unspecified.

WAIT Function

Causes the program that calls it to stop executing until a specified function returns non-NIL. The first argument to WAIT is a reason for waiting, typically a string. The second argument is a function; arguments to the function can be provided as additional arguments to WAIT.

A program that calls the WAIT function stops executing. The function specified in WAIT's second argument is called occasionally, typically when interrupts occur, with the arguments provided in the WAIT call. If the function returns NIL, the program continues to wait. When the function returns non-NIL, WAIT returns an undefined value, and program execution continues.

The testing function you specify with WAIT does not execute in the context of the program that issued the WAIT. Therefore, the testing function cannot depend on the binding of special variables. You should pass the testing function some data structure, such as a cons cell, structure, or array. Pass the same data structure to an interrupt function that modifies the data structure. Chapter 7 in the *VAX LISP/VMS System Access Guide* contains examples of this technique.

For efficiency and reliability, ensure that the testing function executes quickly and does not cause errors. If the testing function encounters an error while LISP is in a WAIT state, LISP is left in an inconsistent state and may have to be terminated. For this reason, WAIT calls its testing function once before entering the WAIT state. Errors that occur on this initial call can be debugged normally.

Format

WAIT *reason function* **&REST** *args*

Arguments

reason
The reason for the wait, typically a string.

function
A function that will be called occasionally to determine if the program should continue to wait.

args
Arguments to be supplied to the function given in the second argument.

Return Value

Unspecified.

Examples

1.
```
Lisp> (setf *flag* (list nil))
(NIL)
Lisp> (bind-keyboard-function
        #\^f
        #'(lambda () (setf (car *flag*) t)))
Lisp> (wait "Wait for Ctrl/F" #'car *flag*)
(After a pause, user types Ctrl/F)
T
Lisp>
```

   - The special variable *FLAG* is set to a list whose only element is NIL.
   - Ctrl/F is bound to a function that sets the first element of *FLAG* to T.
   - The call to the WAIT function specifies CAR as the testing function and *FLAG* as the argument to the testing function. WAIT does not return immediately.
   - When the user types Ctrl/F, the keyboard function sets the first element of *FLAG* to T, the testing function returns T, and the call to WAIT returns.

2.
```
Lisp> (defun set-timer-and-wait (seconds)
        (let* ((delta 0)
               (flag (list nil))
               (iif (instate-interrupt-function
                        #'set-flag
                        :once-only-p t
                        :arguments (list flag))))
          (call-out sys$bintim (time-string seconds) delta)
          (call-out sys$setimr nil delta
                               common-ast-address iif)
          (wait "Timer wait" #'CAR FLAG))
        (princ "The timer has expired")
        t)
SET-TIMER-AND-WAIT
Lisp> (defun set-flag (flag)
        (setf (car flag) t))
SET-FLAG
Lisp> (set-timer-and-wait 5)
(Five seconds elapse) The timer has expired
T
Lisp>
```

   This example uses the definitions of the external routines SYS$SETIMR and SYS$BINTIM and the function TIME-STRING from Examples 1 and 2 under INSTATE-INTERRUPT-FUNCTION.

   - The function SET-TIMER-AND-WAIT is defined. It binds the symbol FLAG to a list whose only element is NIL, then causes that list to be passed to the interrupt function SET-FLAG as its only argument. SET-TIMER-AND-WAIT then calls out to the external routine SYS$SETIMR, specifying that the interrupt function SET-FLAG be executed when the timer expires. Finally, SET-TIMER-AND-WAIT calls the WAIT function, specifying CAR as the testing function and the list to which FLAG is bound as the argument to CAR.

- The function SET-FLAG is defined. It sets the first element of the list passed to it to T.
- SET-TIMER-AND-WAIT is called. It executes as far as the WAIT function call. WAIT does not return until the timer expires and causes the first element of FLAG to be set to T.

WARN Function

Invokes the VAX LISP error handler. The error handler displays an error message and checks the value of the *BREAK-ON-WARNINGS* variable. If the value is NIL, the WARN function returns NIL; if the value is not NIL, the error handler checks the value of the *ERROR-ACTION* variable. The value of the *ERROR-ACTION* variable can be either the :EXIT or the :DEBUG keyword. If the value is :EXIT, the error handler causes the LISP system to exit; if the value is :DEBUG, the handler invokes the VAX LISP Debugger.

For more information on warnings, see *VAX LISP Implementation and Extensions to Common LISP*.

Format

WARN *format-string* **&REST** *args*

Arguments

format-string
The string of characters that is passed to the FORMAT function to create a warning message.

args
The arguments that are passed to the FORMAT function as arguments for the format string.

Return Value

NIL.

Example

```
Lisp> (defun log-error-status (vms-status)
        (declare (special *error-log*))
        (let ((message (get-vms-message vms-status #*1111)))
        (if message
            (write-line message *error-log*)
            (warn
             "There is no message for VMS status #X~8,'0X."
                  vms-status))))
LOG-ERROR-STATUS
```

Defines a function that is an error-logging facility. The function logs the VMS status that is returned from a callout to a system service or an RTL routine. If the callout facility returns an error status that has no corresponding message text, a warning message is displayed, and no log entry is produced.

WITH-GENERALIZED-PRINT-FUNCTION Macro

Locally enables a generalized print function when it evaluates the specified forms. See *VAX LISP Implementation and Extensions to Common LISP* for more information about using generalized print functions.

Format

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

Arguments

name
A symbol identifying the generalized print function that is to be enabled. The enabled generalized print function supersedes any previously enabled generalized print function for *name*.

forms
A call or calls to print functions.

Return Value

Output that is generated by the call or calls to print functions.

Example

```
Lisp> (define-generalized-print-function print-nil-as-list
                (object stream)
                (null object)
           (princ "( )" stream))
PRINT-NIL-AS-LIST
Lisp> (with-generalized-print-function 'print-nil-as-list
          (pprint nil))
( )
```

The PPRINT call prints ( ), because the generalized print function is enabled locally and pretty-printing is enabled.

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua NH 03061
Rest of U.S.A. and Puerto Rico[1]	800–DIGITAL	

[1] Prepaid orders from Puerto Rico, call Digital's local subsidiary (809–754–7575)

From	Call	Write
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order Desk
Internal orders (for software documentation)	—	Software Supply Business (SSB) Digital Equipment Corporation Westminster MA 01473
Internal orders (for hardware documentation)	DTN: 234–4323 508–351–4323	Publishing & Circulation Services (P&CS) NRO3–1/W3 Digital Equipment Corporation Northboro MA 01532

Reader's Comments

VAX LISP/VMS Object Reference Manual
AA–MK72A–TE

Your comments and suggestions will help us improve the quality of our future documentation. Please note that this form is for comments on documentation only.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (product works as described)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

What I like best about this manual: ______________________________

____

What I like least about this manual: ______________________________

____

My additional comments or suggestions for improving this manual:

____

____

I found the following errors in this manual:
Page Description

_____ ______________________________________

_____ ______________________________________

_____ ______________________________________

Please indicate the type of user/reader that you most nearly represent:

□ Administrative Support
□ Computer Operator
□ Educator/Trainer
□ Programmer/Analyst
□ Sales
□ Scientist/Engineer
□ Software Support
□ System Manager
□ Other (please specify) ______________

Name/Title ______________________________ Dept. ______________

Company ______________________________ Date ______________

Mailing Address ______________________________________

______________________________ Phone ______________

10/87

— Do Not Tear — Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
PKO3-1/30D
129 PARKER STREET
MAYNARD, MA 01754-2198

— — Do Not Tear — Fold Here

Cut Along Dotted Line