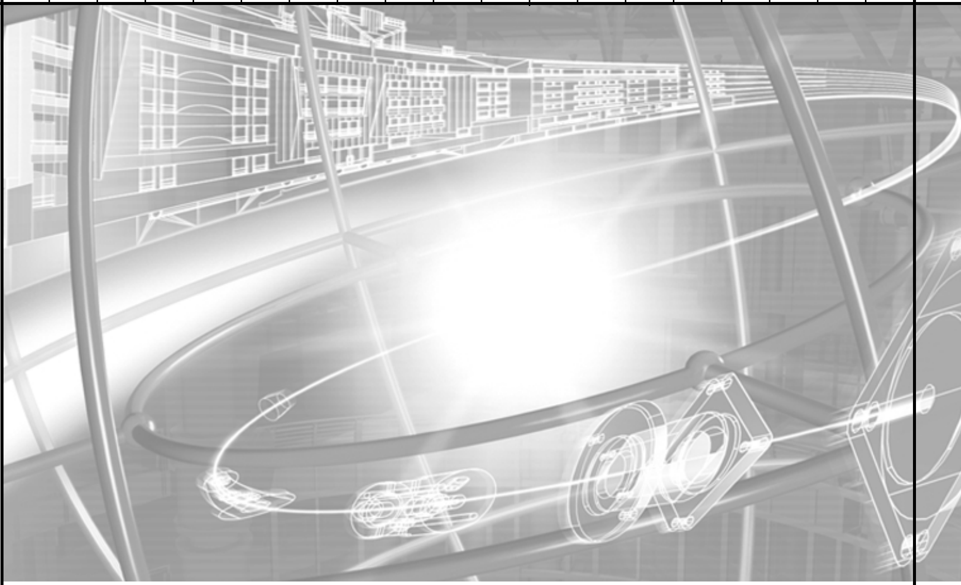
autodesk®

AutoCAD® 2000i



VISUAL LISP REFERENCE GUIDE

1 2 3 4 5 6 7 8 9 10

Contents

AutoLISP Functions

The following is a catalog of the AutoLISP® functions available in AutoCAD. The functions are listed alphabetically.

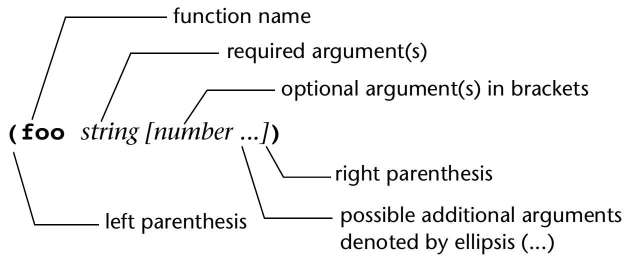
In this chapter, each listing contains a brief description of the function's use and a function syntax statement showing the order and the type of arguments required by the function.

Note that any functions, variables, or features not described here or in other parts of the documentation are not officially supported and are subject to change in future releases.

In this chapter

- Alphabetical listing of AutoLISP functions available in AutoCAD

The following diagram illustrates the format of the function syntax statements in this chapter:


function name
required argument(s)
optional argument(s) in brackets
(foo string [number ...])
right parenthesis
left parenthesis
possible additional arguments denoted by ellipsis (...)

The *number* argument needs additional information: a number can be a real number, an integer, or a symbol set to a real or integer value. If all arguments are integers, the result is an integer. If any of the arguments are real numbers, the integers are promoted to real numbers and the result is a real number.

For more information on syntax statements, see AutoLISP Function Syntax in the *Visual LISP Developer's Guide*.

Note that the value returned by some functions is categorized as *unspecified*. This indicates you cannot rely on using the value returned from this function.

+ (add)

Returns the sum of all numbers

```
(+ [number number] ...)
```

Arguments

number	A number.

Return Values

The result of the addition. If you supply only one *number* argument, this function returns the result of adding it to zero. If you supply no arguments, the function returns 0.

Examples

```
(+ 1 2)              returns  3
(+ 1 2 3 4.5)        returns  10.5
(+ 1 2 3 4.0)        returns  10.0
```

– (subtract)

Subtracts the second and following numbers from the first and returns the difference

```
(– [number number] ...)
```

Arguments

number	A number.

Return Values

The result of the subtraction. If you supply more than two *number* arguments, this function returns the result of subtracting the sum of the second through the last numbers from the first number. If you supply only one *number* argument, this function subtracts the number from zero, and returns a negative number. Supplying no arguments returns 0.

Examples

```
(- 50 40)              returns  10
(- 50 40.0)            returns  10.0
(- 50 40.0 2.5)        returns  7.5
(- 8)                  returns  -8
```

* (multiply)

Returns the product of all numbers

```
(* [number number] ...)
```

Arguments

number	A number.

Return Values

The result of the multiplication. If you supply only one *number* argument, this function returns the result of multiplying it by one; it returns the number. Supplying no arguments returns 0.

Examples

`(* 2 3)`	*returns* `6`
`(* 2 3.0)`	*returns* `6.0`
`(* 2 3 4.0)`	*returns* `24.0`
`(* 3 -4.5)`	*returns* `-13.5`
`(* 3)`	*returns* `3`

/ (divide)

Divides the first number by the product of the remaining numbers and returns the quotient

(/ [number number] ...)

Arguments

number	A number.

Return Values

The result of the division. If you supply more than two *number* arguments, this function divides the first number by the product of the second through the last numbers, and returns the final quotient. If you supply one *number* argument, this function returns the result of dividing it by one; it returns the number. Supplying no arguments returns 0.

Examples

`(/ 100 2)`	*returns* `50`
`(/ 100 2.0)`	*returns* `50.0`
`(/ 100 20.0 2)`	*returns* `2.5`
`(/ 100 20 2)`	*returns* `2`
`(/ 4)`	*returns* `4`

= (equal to)

Compares arguments for numerical equality

(= numstr [numstr] ...)

Arguments

numstr	A number or a string.

Return Values

`T`, if all arguments are numerically equal, `nil` otherwise. If only one argument is supplied, `=` returns `T`.

Examples

```
(= 4 4.0)                 returns  T
(= 20 388)                returns  nil
(= 2.4 2.4 2.4)           returns  T
(= 499 499 500)           returns  nil
(= "me" "me")             returns  T
(= "me" "you")            returns  nil
```

See Also

The eq and equal functions.

/= (not equal to)

Compares arguments for numerical inequality

(/= numstr [numstr] ...)

Arguments

numstr	A number or a string.

Return Values

`T`, if no two successive arguments are the same in value, `nil` otherwise. If only one argument is supplied, `/=` returns `T`.

Note that the behavior of `/=` does not quite conform to other LISP dialects. The standard behavior is to return `T` if no two arguments in the list have the same value. In AutoLISP, `/=` returns `T` if no *successive* arguments have the same value; see the examples that follow.

Examples

```
(/= 10 20)                returns  T
(/= "you" "you")          returns  nil
(/= 5.43 5.44)            returns  T
(/= 10 20 10 20 20)       returns  nil
(/= 10 20 10 20)          returns  T
```

Note in the last example that although there are two arguments in the list with the same value, they do not follow one another, and thus `/=` evaluates to `T`.

< (less than)

Returns `T` if each argument is numerically less than the argument to its right, and returns `nil` otherwise

(< numstr [numstr] ...)

Arguments

numstr	A number or a string.

Return Values

`T`, if each argument is numerically less than the argument to its right, and returns `nil` otherwise. If only one argument is supplied, **<** returns `T`.

Examples

```
(< 10 20)                returns  T
(< "b" "c")              returns  T
(< 357 33.2)             returns  nil
(< 2 3 88)               returns  T
(< 2 3 4 4)              returns  nil
```

<= (less than or equal to)

Returns `T` if each argument is numerically less than or equal to the argument to its right, and returns `nil` otherwise

(<= numstr [numstr] ...)

Arguments

numstr	A number or a string.

Return Values

`T`, if each argument is numerically less than or equal to the argument to its right, and returns `nil` otherwise. If only one argument is supplied, **<=** returns `T`.

Examples

```
(<= 10 20)            returns  T
(<= "b" "b")          returns  T
(<= 357 33.2)         returns  nil
(<= 2 9 9)            returns  T
(<= 2 9 4 5)          returns  nil
```

> (greater than)

Returns `T` if each argument is numerically greater than the argument to its right, and returns `nil` otherwise

`(> numstr [numstr] ...)`

Arguments

numstr	A number or a string.

Return Values

`T`, if each argument is numerically greater than the argument to its right, and `nil` otherwise. If only one argument is supplied, **>** returns `T`.

Examples

```
(> 120 17)            returns  T
(> "c" "b")           returns  T
(> 3.5 1792)          returns  nil
(> 77 4 2)            returns  T
(> 77 4 4)            returns  nil
```

>= (greater than or equal to)

Returns `T` if each argument is numerically greater than or equal to the argument to its right, and returns `nil` otherwise

`(>= numstr [numstr] ...)`

Arguments

numstr	A number or a string.

Return Values

`T`, if each argument is numerically greater than or equal to the argument to its right, and `nil` otherwise. If only one argument is supplied, **>=** returns `T`.

Examples

```
(>= 120 17)               returns   T
(>= "c" "c")              returns   T
(>= 3.5 1792)             returns   nil
(>= 77 4 4)               returns   T
(>= 77 4 9)               returns   nil
```

~ (bitwise NOT)

Returns the bitwise NOT (1's complement) of the argument

`(~ int)`

Arguments

int	An integer.

Return Values

The bitwise NOT (1's complement) of the argument.

Examples

```
(~ 3)                     returns   -4
(~ 100)                   returns   -101
(~ -4)                    returns    3
```

1+ (increment)

Increments a number by 1

`(1+ number)`

Arguments

number	Any number.

Return Values

The argument, increased by 1.

Examples

`(1+ 5)`	*returns*	`6`
`(1+ -17.5)`	*returns*	`-16.5`

1– (decrement)

Decrements a number by 1

`(1– `*`number`*`)`

Arguments

number	Any number.

Return Values

The argument, reduced by 1.

Examples

`(1- 5)`	*returns*	`4`
`(1- -17.5)`	*returns*	`-18.5`

abs

Returns the absolute value of a number

`(abs `*`number`*`)`

Arguments

number	Any number.

Return Values

The absolute value of the argument.

Examples

`(abs 100)`	*returns*	`100`
`(abs -100)`	*returns*	`100`
`(abs -99.25)`	*returns*	`99.25`

acad_colordlg

Displays the standard AutoCAD color selection dialog box

(acad_colordlg *colornum [flag]*)

Arguments

colornum	An integer in the range 0–256 (inclusive), specifying the AutoCAD color number to display as the initial default.
flag	If set to `nil`, disables the ByLayer and ByBlock buttons. Omitting the *flag* argument or setting it to a non-`nil` value enables the ByLayer and ByBlock buttons.

A *colornum* value of 0 defaults to ByBlock, and a value of 256 defaults to ByLayer.

Return Values

The user-selected color number, or `nil`, if the user cancels the dialog box.

Examples

Prompt the user to select a color, and default to green if none is selected:

```
(acad_colordlg 3)
```

acad_helpdlg

Invokes the help facility (obsolete)

(acad_helpdlg *helpfile topic*)

This externally defined function has been replaced by the built-in function **`help`**. It is provided for compatibility with previous releases of AutoCAD.

See Also

The help function for a complete description of this function.

acad_strlsort

Sorts a list of strings by alphabetical order

`(acad_strlsort` *`list`*`)`

Arguments

list	The list of strings to be sorted.

Return Values

The *list* in alphabetical order. If the list is invalid or if there is not enough memory to do the sort, **`acad_strlsort`** returns `nil`.

Examples

Sort a list of abbreviated month names:

```
Command: (setq mos '("Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug"
     "Sep" "Oct" "Nov" "Dec"))
("Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec")

Command: (acad_strlsort mos)
("Apr" "Aug" "Dec" "Feb" "Jan" "Jul" "Jun" "Mar" "May" "Nov" "Oct" "Sep")
```

action_tile

Assigns an action to evaluate when the user selects the specified tile in a dialog box

`(action_tile` *`key action-expression`*`)`

The action assigned by **`action_tile`** supersedes the dialog box's default action (assigned by **`new_dialog`**) or the tile's `action` attribute, if these are specified. The expression can refer to the tile's current value as `$value`, its name as `$key`, its application-specific data (as set by **`client_data_tile`**) as `$data`, its callback reason as `$reason`, and its image coordinates (if the tile is an image button) as `$x` and `$y`.

Arguments

key	A string that names the tile that triggers the action (specified as its `key` attribute). This argument is case-sensitive.
action-expression	A string naming the expression evaluated when the tile is selected.

NOTE You cannot call the AutoLISP **`command`** function from the **`action_tile`** function.

Return Values

`T`

Examples

If `edit1` is a text box, the action expression in the following **`action_tile`** call is evaluated when the user exits the text box:

```
(action_tile "edit1" "(setq ns $value)")
```

See Also

The "Default and DCL Actions" topic in the *Visual LISP Developer's Guide*.

add_list

Adds or modifies a string in the currently active dialog box list

`(add_list` ***`string`*****`)`**

Before using **`add_list`**, you must open the list and initialize it with a call to **`start_list`**. Depending on the operation specified in **`start_list`**, the *string* is either added to the current list or replaces the current list item.

Arguments

string	A string.

Return Values

Returns the string added to the list, if successful, `nil` otherwise.

Examples

Assuming the currently active DCL file has a `popup_list` or `list_box` with a key of `longlist`, the following code fragment initializes the list and adds to it the text strings in `llist`.

```
(setq llist '("first line" "second line" "third line"))
(start_list "longlist")
(mapcar 'add_list llist)
(end_list)
```

After the list has been defined, the following code fragment changes the text in the second line to `"2nd line"`.

```
(start_list "longlist" 1 0)
(add_list "2nd line")
(end_list)
```

See Also

The start_list and end_list functions.

alert

Displays a dialog box containing an error or warning message

`(alert` ***`string`*****`)`**

Arguments

string	The string to appear in the alert box.

Return Values

`nil`

Examples

Display a message in an alert box:

```
(alert "That function is not available.")
```

Display a multiple line message, by using the newline character in *string*:

```
(alert "That function\nis not available.")
```

NOTE Line length and the number of lines in an alert box are platform, device, and window dependent. AutoCAD truncates any string that is too long to fit inside an alert box.

alloc

Sets the size of the segment to be used by the `expand` function

(alloc n-alloc)

Arguments

n-alloc	An integer indicating the amount of memory to be allocated. The integer represents the number of symbols, strings, usubrs, reals, and cons cells.

Return Values

The previous setting of *n-alloc*.

Examples

```
_$ (alloc 100)
1000
```

See Also

The expand function.

and

Returns the logical AND of the supplied arguments

(and *[expr ...]*)

Arguments

expr	Any expression.

Return Values

`Nil`, if any of the expressions evaluate to `nil`, otherwise `T`. If **`and`** is issued without arguments, it returns `T`.

Examples

Command: **(setq a 103 b nil c "string")**
"string"

Command: **(and 1.4 a c)**
T

Command: **(and 1.4 a b c)**
nil

angle

Returns an angle in radians of a line defined by two endpoints

```
(angle pt1 pt2)
```

Arguments

pt1	An endpoint.
pt2	An endpoint.

Return Values

An angle, in radians.

The angle is measured from the *X* axis of the current construction plane, in radians, with angles increasing in the counterclockwise direction. If 3D points are supplied, they are projected onto the current construction plane.

Examples

Command: **(angle '(1.0 1.0) '(1.0 4.0))**
1.5708

Command: **(angle '(5.0 1.33) '(2.4 1.33))**
3.14159

See Also

The “Angular Conversion” topic in the *Visual LISP Developer’s Guide*.

angtof

Converts a string representing an angle into a real (floating-point) value in radians

```
(angtof string [units])
```

Arguments

string	A string describing an angle based on the format specified by the *mode* argument. The *string* must be a string that **angtof** can parse correctly to the specified *unit*. It can be in the same form that **angtos** returns, or in a form that AutoCAD allows for keyboard entry.
units	Specifies the units in which the string is formatted. The value should correspond to values allowed for the AutoCAD system variable AUNITS. If *unit* is omitted, **angtof** uses the current value of AUNITS. The following *unit*s may be specified:

0	Degrees
1	Degrees/minutes/seconds
2	Grads
3	Radians
4	Surveyor's units

Return Values

A real value, if successful, otherwise `nil`.

The **angtof** and **angtos** functions are complementary: if you pass **angtof** a string created by **angtos**, **angtof** is guaranteed to return a valid value, and vice versa (assuming the *unit* values match).

Examples

```
Command: (angtof "45.0000")
0.785398

Command: (angtof "45.0000" 3)
1.0177
```

See Also

The angtos function.

angtos

Converts an angular value in radians into a string

```
(angtos angle [unit [precision]])
```

Arguments

angle	A real number, in radians.
unit	An integer that specifies the angular units. If *unit* is omitted, **angtos** uses the current value of the AutoCAD system variable AUNITS. The following *unit*s may be specified:
	0 Degrees
	1 Degrees/minutes/seconds
	2 Grads
	3 Radians
	4 Surveyor's units
precision	An integer specifying the number of decimal places of precision to be returned. If omitted, **angtos** uses the current setting of the AutoCAD system variable AUPREC.

The **angtos** function takes *angle* and returns it edited into a string according to the settings of *unit*, *precision*, the AutoCAD UNITMODE system variable, and the DIMZIN dimensioning variable.

The **angtos** function accepts a negative *angle* argument, but always reduces it to a positive value between zero and 2 pi radians before performing the specified conversion.

The UNITMODE system variable affects the returned string when surveyor's units are selected (a *unit* value of 4). If UNITMODE = 0, spaces are included in the string (for example, "N 45d E"); if UNITMODE = 1, no spaces are included in the string (for example, "N45dE").

Return Values

A string, if successful, otherwise `nil`.

Examples

Command: **(angtos 0.785398 0 4)**
"45.0000"

Command: **(angtos -0.785398 0 4)**
"315.0000"

Command: **(angtos -0.785398 4)**
"S 45d E"

NOTE Routines that use the **`angtos`** function to display arbitrary angles (those not relative to the value of ANGBASE) should check and consider the value of ANGBASE.

See Also

The angtof function, and "String Conversions" in the *Visual LISP Developer's Guide*.

append

Takes any number of lists and appends them together as one list

`(append [list ...])`

Arguments

list	A list.

Return Values

A list with all arguments appended to the original. If no arguments are supplied, **`append`** returns `nil`.

Examples

Command: **(append '(a b) '(c d))**
(A B C D)

Command: **(append '((a)(b)) '((c)(d)))**
((A) (B) (C) (D))

apply

Passes a list of arguments to, and executes, a specified function

```
(apply 'function list)
```

Arguments

'function	A function. The *function* argument can be either a symbol identifying a **`defun`**, or a **`lambda`** expression.
list	A list. Can be `nil`, if the function accepts no arguments.

Return Values

The result of the function call.

Examples

```
Command: (apply '+ '(1 2 3))
6

Command: (apply 'strcat '("a" "b" "c"))
"abc"
```

arx

Returns a list of the currently loaded ObjectARX applications

```
(arx)
```

Return Values

A list of ObjectARX application file names; the path is not included in the file name.

Examples

```
Command: (arx)
("acadapp.arx" "acmted.arx" "oleaprot.arx")
```

See Also

The arxload and arxunload functions.

arxload

Loads an ObjectARX application

(arxload *application [onfailure]*)

Arguments

application	A quoted string or a variable that contains the name of an executable file. You can omit the *.arx* extension from the file name. You must supply the full path name of the ObjectARX executable file, unless the file is in a directory that is in the AutoCAD Support File Search Path.
onfailure	An expression to be executed if the load fails.

Return Values

The application name, if successful. If unsuccessful and the *onfailure* argument is supplied, **`arxload`** returns the value of this argument, otherwise, failure results in an error message.

If you attempt to load an application that is already loaded, **`arxload`** issues an error message. You may want to check the currently loaded ObjectARX applications with the **`arx`** function before using **`arxload`**.

Examples

Load the *acbrowse.arx* file supplied in the AutoCAD install directory:

```
Command: (arxload "c:/program files/autocad 2000i/acbrowse.arx")
"c:/program files/autocad 2000i/acbrowse.arx"
```

See Also

The arxunload function.

arxunload

Unloads an ObjectARX application

```
(arxunload application [onfailure])
```

Arguments

application	A quoted string or a variable that contains the name of a file that was loaded with the **`arxload`** function. You can omit the *.arx* extension and the path from the file name.
onfailure	An expression to be executed if the unload fails.

Return Values

The application name, if successful. If unsuccessful and the *onfailure* argument is supplied, **`arxunload`** returns the value of this argument, otherwise, failure results in an error message.

Note that locked ObjectARX applications cannot be unloaded. ObjectARX applications are locked by default.

Examples

Unload the acbrowse application that was loaded in the arxload function example:

```
Command: (arxunload "acbrowse")
"acbrowse"
```

See Also

The arxload function.

ascii

Returns the conversion of the first character of a string into its ASCII character code (an integer)

```
(ascii string)
```

Arguments

string	A string.

Return Values

An integer.

Examples

Command: **(ascii "A")**
65

Command: **(ascii "a")**
97

Command: **(ascii "BIG")**
66

assoc

Searches an association list for an element and returns that association list entry

```
(assoc element alist)
```

Arguments

element	Key of an element in an association list.
alist	An association list to be searched.

Return Values

The *alist* entry, if successful. If **`assoc`** does not find *element* as a key in *alist*, it returns `nil`.

Examples

Command: **(setq al '((name box) (width 3) (size 4.7263) (depth 5)))**
((NAME BOX) (WIDTH 3) (SIZE 4.7263) (DEPTH 5))

Command: **(assoc 'size al)**
(SIZE 4.7263)

Command: **(assoc 'weight al)**
nil

atan

Returns the arctangent of a number in radians

`(atan` ***`num1 [num2]`*** **`)`**

Arguments

num1	A number.
num2	A number.

Return Values

The arctangent of *num1*, in radians, if only *num1* is supplied. If you supply both *num1* and *num2* arguments, **`atan`** returns the arctangent of *num1*/*num2*, in radians. If *num2* is zero, it returns an angle of plus or minus 1.570796 radians (+90 degrees or –90 degrees), depending on the sign of *num1*. The range of angles returned is –pi/2 to +pi/2 radians.

Examples

Command: **(atan 1)**
0.785398

Command: **(atan 1.0)**
0.785398

Command: **(atan 0.5)**
0.463648

Command: **(atan 1.0)**
0.785398

Command: **(atan -1.0)**
-0.785398

Command: **(atan 2.0 3.0)**
0.588003

Command: **(atan 2.0 -3.0)**
2.55359

Command: **(atan 1.0 0.0)**
1.5708

atof

Converts a string into a real number

`(atof` ***`string`*****`)`**

Arguments

string	A string to be converted into a real number.

Return Values

A real number.

Examples

Command: **(atof "97.1")**
97.1

Command: **(atof "3")**
3.0

Command: **(atof "3.9")**
3.9

atoi

Converts a string into an integer

```
(atoi string)
```

Arguments

string	A string to be converted into an integer.

Return Values

An integer.

Examples

Command: **(atoi "97")**
97

Command: **(atoi "3")**
3

Command: **(atoi "3.9")**
3

See Also

The itoa function.

atom

Verifies that an item is an atom

(atom *item*)

Arguments

item	Any AutoLISP element.

Some versions of LISP differ in their interpretation of **atom**, so be careful when converting from non-AutoLISP code.

Return Values

`Nil` if *item* is a list, otherwise `T`. Anything that is not a list is considered an atom.

Examples

```
Command: (setq a '(x y z))
(X Y Z)

Command: (setq b 'a)
A

Command: (atom 'a)
T

Command: (atom a)
nil

Command: (atom 'b)
T

Command: (atom b)
T

Command: (atom '(a b c))
nil
```

atoms-family

Returns a list of the currently defined symbols

`(atoms-family` ***`format [symlist]`*** **`)`**

Arguments

format	An integer value of 0 or 1 that determines the format in which **`atoms-family`** returns the symbol names: **0** Return the symbol names as a list **1** Return the symbol names as a list of strings
symlist	A list of strings that specify the symbol names you want **`atoms-family`** to search for.

Return Values

A list of symbols. If you specify **`symlist`**, then **`atoms-family`** returns the specified symbols that are currently defined, and returns `nil` for those symbols that are not defined.

Examples

Command: **(atoms-family 0)**
(BNS_PRE_SEL FITSTR2LEN C:AI_SPHERE ALERT DEFUN C:BEXTEND REM_GROUP
B_RESTORE_SYSVARS BNS_CMD_EXIT LISPED FNSPLITL...

The following code verifies that the symbols `CAR`, `CDR`, and `XYZ` are defined, and returns the list as strings:

Command: **(atoms-family 1 '("CAR" "CDR" "XYZ"))**
("CAR" "CDR" nil)

The return value shows that the symbol `XYZ` is not defined.

autoarxload

Predefines command names to load an associated ObjectARX file

`(autoarxload` *`filename cmdlist`*`)`

The first time a user enters a command specified in *cmdlist*, AutoCAD loads the ObjectARX application specified in *filename*, then continues the command.

If you associate a command with *filename* and that command is not defined in the specified file, AutoCAD alerts you with an error message when you enter the command.

Arguments

filename	A string specifying the *.arx* file to be loaded when one of the commands defined by the *cmdlist* argument is entered at the Command prompt. If you omit the path from *filename*, AutoCAD looks for the file in the Support File Search Path.
cmdlist	A list of strings.

Return Values

`nil`

Examples

The following code defines the `C:APP1`, `C:APP2`, and `C:APP3` functions to load the *bonusapp.arx* file:

```
(autoarxload "BONUSAPP" '("APP1" "APP2" "APP3"))
```

autoload

Predefines command names to load an associated AutoLISP file

`(autoload` *`filename cmdlist`*`)`

The first time a user enters a command specified in *cmdlist*, AutoCAD loads the application specified in *filename*, then continues the command.

Arguments

filename	A string specifying the *.lsp* file to be loaded when one of the commands defined by the *cmdlist* argument is entered at the Command prompt. If you omit the path from *filename*, AutoCAD looks for the file in the Support File Search Path.
cmdlist	A list of strings.

Return Values

`nil`

If you associate a command with *filename* and that command is not defined in the specified file, AutoCAD alerts you with an error message when you enter the command.

Examples

The following causes AutoCAD to load the *bonusapp.lsp* file the first time the APP1, APP2, or APP3 commands are entered at the Command prompt:

```
(autoload "BONUSAPP" '("APP1" "APP2" "APP3"))
```

Boole

Serves as a general bitwise Boolean function

`(Boole operator int1 [int2 ...])`

Arguments

operator	An integer between 0 and 15 representing one of the 16 possible Boolean functions in two variables.
int1, int2...	Integers.
	Note that **`Boole`** will accept a single integer argument, but the result is unpredictable.

Successive integer arguments are bitwise (logically) combined based on this function and on the following truth table:

Boolean truth table

Int1	Int2	operator bit
0	0	8
0	1	4
1	0	2
1	1	1

Each bit of *int1* is paired with the corresponding bit of *int2*, specifying one horizontal row of the truth table. The resulting bit is either 0 or 1, depending on the setting of the *operator* bit that corresponds to this row of the truth table.

If the appropriate bit is set in *operator*, the resulting bit is 1; otherwise the resulting bit is 0. Some of the values for *operator* are equivalent to the standard Boolean operations AND, OR, XOR, and NOR.

Boole function bit values

Operator	Operation	Resulting bit is 1 if
1	AND	Both input bits are 1
6	XOR	Only one of the two input bits is 1
7	OR	Either or both of the input bits are 1
8	NOR	Both input bits are 0 (1's complement)

Return Values

An integer.

Examples

The following specifies a logical AND of the values 12 and 5:

```
Command: (Boole 1 12 5)
4
```

The following specifies a logical XOR of the values 6 and 5:

Command: **(Boole 6 6 5)**
3

You can use other values of *operator* to perform other Boolean operations for which there are no standard names. For example, if *operator* is 4, the resulting bits are set if the corresponding bits are set in *int2* but not in *int1*:

Command: **(Boole 4 3 14)**
12

boundp

Verifies if a value is bound to a symbol

`(boundp` *`sym`*`)`

Arguments

sym	A symbol.

Return Values

`T` if *sym* has a value bound to it. If no value is bound to *sym*, or if it has been bound to `nil`, **`boundp`** returns `nil`. If *sym* is an undefined symbol, it is automatically created and is bound to `nil`.

Examples

Command: **(setq a 2 b nil)**
nil

Command: **(boundp 'a)**
T

Command: **(boundp 'b)**
nil

The **`atoms-family`** function provides an alternative method of determining the existence of a symbol without automatically creating the symbol.

See Also

The atoms-family function.

caddr

Returns the third element of a list

```
(caddr list)
```

In AutoLISP, `caddr` is frequently used to obtain the *Z* coordinate of a 3D point (the third element of a list of three reals).

Arguments

list	A list.

Return Values

The third element in *list;* or `nil`, if the list is empty or contains fewer than three elements.

Examples

```
Command: (setq pt3 '(5.25 1.0 3.0))
(5.25 1.0 3.0)

Command: (caddr pt3)
3.0

Command: (caddr '(5.25 1.0))
nil
```

See Also

The “Point Lists” topic in the *Visual LISP Developer’s Guide*.

cadr

Returns the second element of a list

```
(cadr list)
```

In AutoLISP, `cadr` is frequently used to obtain the *Y* coordinate of a 2D or 3D point (the second element of a list of two or three reals).

Arguments

list	A list.

Return Values

The second element in *list,* or `nil`, if the list is empty or contains only one element.

Examples

Command: **(setq pt2 '(5.25 1.0))**
(5.25 1.0)

Command: **(cadr pt2)**
1.0

Command: **(cadr '(4.0))**
nil

Command: **(cadr '(5.25 1.0 3.0))**
1.0

See Also

The "Point Lists" topic in the *Visual LISP Developer's Guide*.

car

Returns the first element of a list

```
(car list)
```

Arguments

list	A list.

Return Values

The first element in *list;* or `nil`, if the list is empty.

Examples

Command: **(car '(a b c))**
A

Command: **(car '((a b) c))**
(A B)

Command: **(car '())**
nil

See Also

The “Point Lists” topic in the *Visual LISP Developer’s Guide*.

cdr

Returns a list containing all but the first element of the specified list

```
(cdr list)
```

Arguments

list	A list.

Return Values

A list containing all the elements of *list,* except the first element (but see Note below). If the list is empty, **`cdr`** returns `nil`.

NOTE When the *list* argument is a dotted pair, **`cdr`** returns the second element without enclosing it in a list.

Examples

```
Command: (cdr '(a b c))
(B C)

Command: (cdr '((a b) c))
(C)

Command: (cdr '())
nil

Command: (cdr '(a . b))
B

Command: (cdr '(1 . "Text"))
"Text"
```

See Also

The “Point Lists” topic in the *Visual LISP Developer’s Guide*.

chr

Converts an integer representing an ASCII character code into a single-character string

`(chr integer)`

Arguments

list	An integer.

Return Values

A string containing the ASCII character code for *integer*. If the integer is not in the range of 1–255, the return value is unpredictable.

Examples

Command: **(chr 65)**
"A"

Command: **(chr 66)**
"B"

Command: **(chr 97)**
"a"

client_data_tile

Associates application-managed data with a dialog box tile

`(client_data_tile key clientdata)`

Arguments

key	A string that specifies a tile. This argument is case-sensitive.
clientdata	A string to be associated with the *key* tile. An action expression or callback function can refer to the string as `$data`.

Return Values

`nil`

close

Closes an open file

(close *file-desc*)

Arguments

file-desc	A file descriptor obtained from the **open** function.

Return Values

`Nil` if *file-desc* is valid, otherwise results in an error message.

After a **close**, the file descriptor is unchanged but is no longer valid. Data added to an open file is not actually written until the file is closed.

Examples

The following code counts the number of lines in the file *somefile.txt* and sets the variable `ct` equal to that number:

```
(setq fil "SOMEFILE.TXT")
(setq x (open fil "r") ct 0)
(while (read-line x)
  (setq ct (1+ ct))
)
(close x)
```

command

Executes an AutoCAD command

(command *[arguments]* ...)

Arguments

arguments	AutoCAD commands and their options.
	The *arguments* to the **command** function can be strings, reals, integers, or points, as expected by the prompt sequence of the executed command. A null string (`""`) is equivalent to pressing ENTER on the keyboard. Invoking **command** with no argument is equivalent to pressing ESC and cancels most AutoCAD commands.

The **command** function evaluates each argument and sends it to AutoCAD in response to successive prompts. It submits command names and options as strings, 2D points as lists of two reals, and 3D points as lists of three reals. AutoCAD recognizes command names only when it issues a Command prompt.

Note that if you issue **command** from Visual LISP, focus does not change to the AutoCAD window. If the command requires user input, you'll see the return value (`nil`) in the Console window, but AutoCAD will be waiting for input. You must manually activate the AutoCAD window and respond to the prompts. Until you do so, any subsequent commands will fail.

Return Values

`nil`

Examples

The following example sets two variables `pt1` and `pt2` equal to two point values 1,1 and 1,5. It then uses the **command** function to issue the LINE command and pass the two point values.

```
Command: (setq pt1 '(1 1) pt2 '(1 5))
(1 5)

Command: (command "line" pt1 pt2 "")
line From point:
To point:
To point:
Command: nil
```

Restrictions and Notes

The AutoCAD SKETCH command reads the digitizer directly and therefore cannot be used with the AutoLISP **command** function. If the SCRIPT command is used with the **command** function, it should be the last function call in the AutoLISP routine.

Also, if you use the **command** function in an *acad.lsp* or *.mnl* file, it should be called only from within a **defun** statement. Use the **S::STARTUP** function to define commands that need to be issued immediately when you begin a drawing session.

For AutoCAD commands that require the selection of an object (like the BREAK and TRIM commands), you can supply a list obtained with **entsel** instead of a point to select the object. For examples, see "Passing Pick Points to AutoCAD Commands" in the *Visual LISP Developer's Guide*.

Commands executed from the **command** function are not echoed to the command line if the CMDECHO system variable (accessible from **setvar** and **getvar**) is set to 0.

See Also

The vl-cmdf function in this reference and "Command Submission" in the *Visual LISP Developer's Guide*.

cond

Serves as the primary conditional function for AutoLISP

(cond *[(test result ...) ...]*)

The **cond** function accepts any number of lists as arguments. It evaluates the first item in each list (in the order supplied) until one of these items returns a value other than `nil`. It then evaluates those expressions that follow the test that succeeded.

Return Values

The value of the last expression in the sublist. If there is only one expression in the sublist (that is, if *result* is missing), the value of the *test* expression is returned. If no arguments are supplied, **cond** returns `nil`.

Examples

The following example uses **cond** to perform an absolute value calculation:

```
(cond
   ((minusp a) (- a))
   (t a)
)
```

If the variable `a` is set to the value –10, this returns 10.

As shown, **cond** can be used as a *case* type function. It is common to use `T` as the last (default) *test* expression. Here's another simple example. Given a user response string in the variable `s`, this function tests the response and returns 1 if it is `Y` or `y`, 0 if it is `N` or `n`, and `nil` otherwise.

```
(cond
   ((= s "Y") 1)
   ((= s "y") 1)
   ((= s "N") 0)
   ((= s "n") 0)
   (t nil)
)
```

cons

Adds an element to the beginning of a list, or constructs a dotted list

```
(cons new-first-element list-or-atom)
```

Arguments

new-first-element	Element to be added to the beginning of a list. This element can be an atom or a list.
list-or-atom	A list or an atom.

Return Values

The value returned depends on the data type of *list-or-atom*. If *list-or-atom* is a list, **cons** returns that list with *new-first-element* added as the first item in the list. If *list-or-atom* is an atom, **cons** returns a dotted pair consisting of *new-first-element* and *list-or-atom*.

Examples

```
Command: (cons 'a '(b c d))
(A B C D)
```

```
Command: (cons '(a) '(b c d))
((A) B C D)
```

```
Command: (cons 'a 2)
(A . 2)
```

See Also

The “List Handling” topic in the *Visual LISP Developer’s Guide*.

cos

Returns the cosine of an angle expressed in radians

(cos *ang*)

Arguments

ang	An angle, in radians.

Return Values

The cosine of *ang*, in radians.

Examples

Command: **(cos 0.0)**
1.0

Command: **(cos pi)**
-1.0

cvunit

Converts a value from one unit of measurement to another

(cvunit *value from-unit to-unit*)

Arguments

value	The numeric value or point list (2D or 3D point) to be converted.
from-unit	The unit that *value* is being converted from.
to-unit	The unit that *value* is being converted to.

The *from-unit* and *to-unit* arguments can name any unit type found in the *acad.unt* file.

Return Values

The converted value, if successful, or `nil`, if either unit name is unknown (not found in the *acad.unt* file), or if the two units are incompatible (for example, trying to convert grams into years).

Examples

Command: **(cvunit 1 "minute" "second")**
60.0

Command: **(cvunit 1 "gallon" "furlong")**
nil

Command: **(cvunit 1.0 "inch" "cm")**
2.54

Command: **(cvunit 1.0 "acre" "sq yard")**
4840.0

Command: **(cvunit '(1.0 2.5) "ft" "in")**
(12.0 30.0)

Command: **(cvunit '(1 2 3) "ft" "in")**
(12.0 24.0 36.0)

NOTE If you have several values to convert in the same manner, it is more efficient to convert the value 1.0 once and then apply the resulting value as a scale factor in your own function or computation. This works for all predefined units except temperature, where an offset is involved as well.

See Also

The “Unit Conversion” topic in the *Visual LISP Developer's Guide*.

defun

Defines a function

(defun *sym* (*[arguments]* *[/ variables...]*) *expr...*)

Arguments

sym	A symbol naming the function.
arguments	The names of arguments expected by the function.
/ variables	The names of one or more local variables for the function. The slash preceding the variable names must be separated from the first local name and from the last argument, if any, by at least one space.

expr	Any number of AutoLISP expressions to be evaluated when the function executes.

If you do not declare any arguments or local symbols, you must supply an empty set of parentheses after the function name.

If duplicate argument or symbol names are specified, AutoLISP uses the first occurrence of each name and ignores the following occurrences.

Return Values

The result of the last expression evaluated.

WARNING! Never use the name of a built-in function or symbol for the *sym* argument to **`defun`**. This overwrites the original definition and makes the built-in function or symbol inaccessible. To get a list of built-in and previously defined functions, use the **`atoms-family`** function.

Examples

`(defun myfunc (x y) ...)`	*Function takes two arguments*
`(defun myfunc (/ a b) ...)`	*Function has two local variables*
`(defun myfunc (x / temp) ...)`	*One argument, one local variable*
`(defun myfunc () ...)`	*No arguments or local variables*

See Also

The "Symbol and Function Handling" topic in the *Visual LISP Developer's Guide*.

defun-q

Defines a function as a list

`(defun-q` ***`sym ([arguments] [/ variables...])`*** **`expr...)`**

The **`defun-q`** function is provided strictly for backward-compatibility with previous versions of AutoLISP, and should not be used for other purposes. You can use **`defun-q`** in situations where you need to access a function definition as a list structure, which is the way **`defun`** was implemented in previous, non-compiled versions of AutoLISP.

Arguments

sym	A symbol naming the function.
arguments	The names of arguments expected by the function.
/ variables	The names of one or more local variables for the function. The slash preceding the variable names must be separated from the first local name and from the last argument, if any, by at least one space.
expr	Any number of AutoLISP expressions to be evaluated when the function executes.

If you do not declare any arguments or local symbols, you must supply an empty set of parentheses after the function name.

If duplicate argument or symbol names are specified, AutoLISP uses the first occurrence of each name and ignores the following occurrences.

Return Values

The result of the last expression evaluated.

Examples

```
_$ (defun-q my-startup (x) (print (list x)))
MY-STARTUP
```

```
_$ (my-startup 5)
(5) (5)
```

Use **`defun-q-list-ref`** to display the list structure of **`my-startup`**:

```
_$ (defun-q-list-ref 'my-startup)
((X) (PRINT (LIST X)))
```

See Also

The defun-q-list-ref and defun-q-list-set functions.

defun-q-list-ref

Displays the list structure of function defined with defun-q

`(defun-q-list-ref '`*`function`*` )`

Arguments

function	A symbol naming the function.

Return Values

The list definition of the function, or `nil`, if the argument is not a list.

Examples

Define a function using **`defun-q`**:

```
_$ (defun-q my-startup (x) (print (list x)))
MY-STARTUP
```

Use **`defun-q-list-ref`** to display the list structure of **`my-startup`**:

```
_$ (defun-q-list-ref 'my-startup)
((X) (PRINT (LIST X)))
```

See Also

The defun-q and defun-q-list-set functions.

defun-q-list-set

Sets the value of a symbol to be a function defined by a list

`(defun-q-list-set '`*`sym list`*`)`

Arguments

sym	A symbol naming the function
list	A list containing the expressions to be included in the function.

Return Values

The *sym* defined.

Examples

```
_$ (defun-q-list-set 'foo '((x) x))
FOO

_$ (foo 3)
3
```

The following example illustrates the use of **`defun-q-list-set`** to combine two functions into a single function. First, from the Visual LISP Console window, define two functions with **`defun-q`**:

```
_$ (defun-q s::startup (x) (print x))
S::STARTUP

_$ (defun-q my-startup (x) (print (list x)))
MY-STARTUP
```

Use **`defun-q-list-set`** to combine the functions into a single function:

```
_$ (defun-q-list-set 's::startup (append
   (defun-q-list-ref 's::startup)
   (cdr (defun-q-list-ref 'my-startup))))
S::STARTUP
```

The following illustrates how the functions respond individually, and how the functions work after being combined using **`defun-q-list-set`**:

```
_$ (defun-q foo (x) (print (list 'foo x)))
FOO

_$ (foo 1)
(FOO 1) (FOO 1)

_$ (defun-q bar (x) (print (list 'bar x)))
BAR

_$ (bar 2)
(BAR 2) (BAR 2)

_$ (defun-q-list-set
  'foo
  (append (defun-q-list-ref 'foo)
              (cdr (defun-q-list-ref 'bar))
  ))
FOO

_$ (foo 3)
(FOO 3)
(BAR 3) (BAR 3)
```

See Also

The defun-q and defun-q-list-ref functions.

dictadd

Adds a nongraphical object to the specified dictionary

```
(dictadd ename symbol newobj)
```

Arguments

ename	Name of the dictionary the object is being added to.
symbol	The key name of the object being added to the dictionary; *symbol* must be a unique name that does not already exist in the dictionary.
newobj	A nongraphical object to be added to the dictionary.

As a general rule, each object added to a dictionary must be unique to that dictionary. This is specifically a problem when adding group objects to the group dictionary. Adding the same group object using different key names results in duplicate group names which can send the **`dictnext`** function into an infinite loop.

Return Values

The entity name of the object added to the dictionary.

Examples

The examples that follow create objects and add them to the named object dictionary.

Create a dictionary entry list:

Command: **(setq dictionary (list '(0 . "DICTIONARY") '(100 . "AcDbDictionary")))**
((0 . "DICTIONARY") (100 . "AcDbDictionary"))

Create a dictionary object using the **`entmakex`** function:

Command: **(setq xname (entmakex dictionary))**
<Entity name: 1d98950>

Add the dictionary to the named object dictionary:

Command: **(setq newdict (dictadd (namedobjdict) "MY_WAY_COOL_DICTIONARY" xname))**
<Entity name: 1d98950>

Create an Xrecord list:

```
Command: (setq datalist (append (list '(0 . "XRECORD")'(100 .
"AcDbXrecord")) '((1 . "This is my data") (10 1. 2. 3.) (70 . 33))))
((0 . "XRECORD") (100 . "AcDbXrecord") (1 . "This is my data") (10 1.0 2.0 3.0)
(70 . 33))
```

Make an Xrecord object:

```
Command: (setq xname (entmakex datalist))
<Entity name: 1d98958>
```

Add the Xrecord object to the dictionary:

```
Command: (dictadd newdict "DATA_RECORD_1" xname)
<Entity name: 1d98958>
```

See Also

The dictnext, dictremove, dictrename, dictsearch, and namedobjdict functions.

dictnext

Finds the next item in a dictionary

`(dictnext` ***`ename [rewind]`*****`)`**

Arguments

ename	Name of the dictionary being viewed.
rewind	If this argument is present and is not `nil`, the dictionary is rewound and the first entry in it is retrieved.

Return Values

The next entry in the specified dictionary, or `nil`, when the end of the dictionary is reached. Entries are returned as lists of dotted pairs of DXF-type codes and values. Deleted dictionary entries are not returned.

The **`dictsearch`** function specifies the initial entry retrieved.

Use **`namedobjdict`** to obtain the master dictionary entity name.

NOTE Once you begin stepping through the contents of a dictionary, passing a different dictionary name to **`dictnext`** will cause the place to be lost in the original dictionary. In other words, only one global iterator is maintained for use in this function.

Examples

Create a dictionary and an entry as shown in the example for **`dictadd`**. Then make another Xrecord object:

Command: **(setq xname (entmakex datalist))**
<Entity name: 1b62d60>

Add this Xrecord object to the dictionary, as the second record in the dictionary:

Command: **(dictadd newdict "DATA_RECORD_2" xname)**
<Entity name: 1b62d60>

Return the entity name of the next entry in the dictionary:

Command: **(cdr (car (dictnext newdict)))**
<Entity name: 1bac958>

`dictnext` returns the name of the first entity added to the dictionary.

Return the entity name of the next entry in the dictionary:

Command: **(cdr (car (dictnext newdict)))**
<Entity name: 1bac960>

`dictnext` returns the name of the second entity added to the dictionary.

Return the entity name of the next entry in the dictionary:

Command: **(cdr (car (dictnext newdict)))**
nil

There are no more entries in the dictionary, so **`dictnext`** returns `nil`.

Rewind to the first entry in the dictionary and return the entity name of that entry:

Command: **(cdr (car (dictnext newdict T)))**
<Entity name: 1bac958>

Specifying `T` for the optional *rewind* argument causes **`dictnext`** to return the first entry in the dictionary.

See Also

The dictadd, dictremove, dictrename, dictsearch, and namedobjdict functions.

dictremove

Removes an entry from the specified dictionary

`(dictremove` ***`ename symbol`*****`)`**

By default, removing an entry from a dictionary does not delete it from the database. This must be done with a call to **`entdel`**. Currently the exceptions to this rule are groups and mlinestyles. The code that implements these features requires that the database and these dictionaries be up to date, and therefore automatically deletes the entity when it is removed (with **`dictremove`**) from the dictionary.

Arguments

ename	Name of the dictionary being modified.
symbol	The entry to be removed from *ename*.

The **`dictremove`** function does not allow the removal of an mlinestyle from the mlinestyle dictionary if it is actively referenced by an mline in the database.

Return Values

The entity name of the removed entry. If *ename* is invalid or *symbol* is not found, **`dictremove`** returns `nil.`

Examples

The following example removes the dictionary created in the **`dictadd`** example:

Command: **(dictremove (namedobjdict) "my_way_cool_dictionary")**
<Entity name: 1d98950>

See Also

The dictadd, dictnext, dictrename, dictsearch, and namedobjdict functions.

dictrename

Renames a dictionary entry

```
(dictrename ename oldsym newsym)
```

Arguments

ename	Name of the dictionary being modified.
oldsym	Original key name of the entry.
newsym	New key name of the entry.

Return Values

The *newsym* value, if the rename is successful. If either the *oldname* is not present in the dictionary, or *ename* is invalid, or *newname* is invalid, or *newname* is already present in the dictionary, **`dictrename`** returns `nil`.

Examples

The following example renames the dictionary created in the **`dictadd`** sample:

Command: **(dictrename (namedobjdict) "my_way_cool_dictionary" "An even cooler dictionary")**
"An even cooler dictionary"

See Also

The dictadd, dictnext, dictremove, dictsearch, and namedobjdict functions.

dictsearch

Searches a dictionary for an item

```
(dictsearch ename symbol [setnext])
```

Arguments

ename	Name of the dictionary being searched.
symbol	A string that specifies the item to be searched for within the dictionary.

setnext	If present and not `nil`, the **`dictnext`** entry counter is adjusted so the following **`dictnext`** call returns the entry after the one returned by this **`dictsearch`** call.

Return Values

The entry for the specified item, if successful, or `nil`, if no entry is found.

Examples

The following example illustrates the use of **`dictsearch`** to obtain the dictionary added in the **`dictadd`** example:

```
Command: (setq newdictlist (dictsearch (namedobjdict)
"my_way_cool_dictionary"))
((-1 . <Entity name: 1d98950>) (0 . "DICTIONARY") (5 . "52") (102 .
"{ACAD_REACTORS") (330 . <Entity name: 1d98860>) (102 . "}") (330 . <Entity
name: 1d98860>) (100 . "AcDbDictionary") (280 . 0) (281 . 1) (3 .
"DATA_RECORD_1") (350 . <Entity name: 1d98958>))
```

See Also

The dictadd, dictnext, dictremove, and namedobjdict functions.

dimx_tile

Retrieves the width of a tile in dialog box units

`(dimx_tile` ***`key`*****`)`**

Arguments

key	A string specifying the tile to be queried. The *key* argument is case-sensitive.

Return Values

The width of the tile.

The coordinates returned are the maximum allowed within the tile. Because coordinates are zero based, this functions return one less than the total *X* dimension (*X*–1). The **`dimx_tile`** and **`dimy_tile`** functions are provided for use with **`vector_image`**, **`fill_image`**, and **`slide_image`**, which require you to specify absolute tile coordinates.

Examples

```
(setq tile_width (dimx_tile "my_tile"))
```

dimy_tile

Retrieves the height of a tile in dialog box units

(dimy_tile *key*)

Arguments

key	A string specifying the tile to be queried. The *key* argument is case-sensitive.

Return Values

The height of the tile.

The coordinates returned are the maximum allowed within the tile. Because coordinates are zero based, this functions return one less than the total *Y* dimension (*Y*–1). The **dimx_tile** and **dimy_tile** functions are provided for use with **vector_image**, **fill_image**, and **slide_image**, which require you to specify absolute tile coordinates.

Examples

```
(setq tile_height (dimy_tile "my_tile"))
```

distance

Returns the 3D distance between two points

(distance *pt1 pt2*)

Arguments

pt1	A 2D or 3D point list.
pt1	A 2D or 3D point list.

Return Values

The distance.

If one or both of the supplied points is a 2D point, then **distance** ignores the *Z* coordinates of any 3D points supplied and returns the 2D distance between the points as projected into the current construction plane.

Examples

Command: **(distance '(1.0 2.5 3.0) '(7.7 2.5 3.0))**
6.7

Command: **(distance '(1.0 2.0 0.5) '(3.0 4.0 0.5))**
2.82843

See Also

The "Geometric Utilities" topic in the *Visual LISP Developer's Guide*.

distof

Converts a string that represents a real (floating-point) value into a real value

```
(distof string [mode])
```

The **distof** and **rtos** functions are complementary. If you pass **distof** a string created by **rtos**, **distof** is guaranteed to return a valid value, and vice versa (assuming the mode values are the same).

Arguments

string	A string to be converted. The argument must be a string that **distof** can parse correctly according to the units specified by *mode*. It can be in the same form that **rtos** returns, or in a form that AutoCAD allows for keyboard entry.
mode	The units in which the string is currently formatted. The *mode* corresponds to the values allowed for the AutoCAD system variable LUNITS. Specify one of the following numbers for *mode*:

1 Scientific

2 Decimal

3 Engineering (feet and decimal inches)

4 Architectural (feet and fractional inches)

5 Fractional

Return Values

A real number, if successful, otherwise `nil`.

NOTE The **`distof`** function treats modes 3 and 4 the same. That is, if *mode* specifies 3 (engineering) or 4 (architectural) units, and *string* is in either of these formats, **`distof`** returns the correct real value.

---

done_dialog

Terminates a dialog box

`(done_dialog` ***`[status]`*** **`)`**

Arguments

status	A positive integer that **`start_dialog`** will return instead of returning `1` for OK or `0` for Cancel. The meaning of any *status* value greater than 1 is determined by your application.

You must call **`done_dialog`** from within an action expression or callback function (see “**`action_tile`**”).

Return Values

A two-dimensional point list that is the (*X*,*Y*) location of the dialog box when the user exited it.

Usage Notes

If you provide a callback for the button whose key is `"accept"` or `"cancel"` (usually the OK and Cancel buttons), the callback must call **`done_dialog`** explicitly. If it doesn’t, the user can be trapped in the dialog box. If you don’t provide an explicit callback for these buttons and use the standard exit buttons, AutoCAD handles them automatically. Also, an explicit AutoLISP action for the `“accept”` button must specify a *status* of 1 (or an application-defined value); otherwise, **`start_dialog`** returns the default value, 0, which makes it appear as if the dialog box was canceled.

end_image

Ends creation of the currently active dialog box image

`(end_image)`

This function is the complement of **`start_image`**.

Return Values

`nil`

See Also

The start_image function.

end_list

Ends processing of the currently active dialog box list

`(end_list)`

This function is the complement of **`start_list`**.

Return Values

`nil`

See Also

The add_list and start_list functions.

entdel

Deletes objects (entities) or restores previously deleted objects

`(entdel` ***`ename`*****`)`**

The entity specified by *ename* is deleted if it is currently in the drawing. The **`entdel`** function restores the entity to the drawing if it has been deleted previously in this editing session. Deleted entities are purged from the drawing when the drawing is exited. The **`entdel`** function can delete both graphical and nongraphical entities.

Arguments

ename	Name of the entity to be deleted or restored.

Return Values

The entity name.

Usage Notes

The **`entdel`** function operates only on main entities. Attributes and polyline vertices cannot be deleted independently of their parent entities. You can use the **`command`** function to operate the ATTEDIT or PEDIT commands to modify subentities.

You cannot delete entities within a block definition. However, you can completely redefine a block definition, minus the entity you want deleted, with **`entmake`**.

Examples

Get the name of the first entity in the drawing and assign it to variable `e1`:

```
Command: (setq e1 (entnext))
<Entity name: 2c90520>
```

Delete the entity named by e1:

```
Command: (entdel e1)
<Entity name: 2c90520>
```

Restore the entity named by e1:

```
Command: (entdel e1)
 <Entity name: 2c90520>
```

entget

Retrieves an object's (entity's) definition data

`(entget` ***`ename [applist]`*** **`)`**

Arguments

ename	Name of the entity being queried. The *ename* can refer to either a graphical or nongraphical entity.
applist	A list of registered application names.

Return Values

An association list containing the entity definition of *ename*. If you specify the optional *applist* argument, **`entget`** also returns the extended data associated with the specified applications. Objects in the list are assigned AutoCAD DXF group codes for each part of the entity data.

Note that the DXF group codes used by AutoLISP differ slightly from the group codes in a DXF file. The AutoLISP DXF group codes are documented in the *DXF Reference*.

Examples

Assume that the last object created in the drawing is a line drawn from point (1,2) to point (6,5). The following example shows code that retrieves the entity name of the last object with the **`entlast`** function, and passes that name to **`entget`**:

```
Command: (entget (entlast))
((-1 . <Entity name: 1bbd1d0>) (0 . "LINE") (330 . <Entity name: 1bbd0c8>) (5 .
"6A") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine")
(10 1.0 2.0 0.0) (11 6.0 5.0 0.0) (210 0.0 0.0 1.0))
```

See Also

The entdel, entlast, entmod, entmake, entnext, and entupd functions. See “Entity Data Functions” in the *Visual LISP Developer’s Guide*.

entlast

Returns the name of the last nondeleted main object (entity) in the drawing

```
(entlast)
```

The **`entlast`** function is frequently used to obtain the name of a new entity that has just been added with the **`command`** function. To be selected, the entity need not be on the screen or on a thawed layer.

Return Values

An entity name, or `nil`, if there are no entities in the current drawing.

Examples

Set variable `e1` to the name of the last entity added to the drawing:

Command: **(setq e1 (entlast))**
<Entity name: 2c90538>

If your application requires the name of the last nondeleted entity (main entity or subentity), define a function such as the following and call it instead of **`entlast`**.

```
(defun lastent (/ a b)
  (if (setq a (entlast))              Gets last main entity
    (while (setq b (entnext a))       If subentities follow, loops
                                      until there are no more
      (setq a b)                      subentities
    )
  )
  a                                   Returns last main entity
)                                     or subentity
```

See Also

The entdel, entget, entmod, entnext, and entsel functions.

entmake

Creates a new entity in the drawing

`(entmake` ***`[elist]`*****`)`**

The **`entmake`** function can define both graphical and nongraphical entities.

Arguments

elist	A list of entity definition data in a format similar to that returned by the **`entget`** function. The *elist* argument must contain all of the information necessary to define the entity. If any required definition data is omitted, **`entmake`** returns `nil` and the entity is rejected. If you omit optional definition data (such as the layer), **`entmake`** uses the default value.

The entity type (for example, `CIRCLE` or `LINE`) must be the first or second field of the *elist*. If entity type is the second field, it can be preceded only by the entity name. The **`entmake`** function ignores the entity name when creating the new entity. If the *elist* contains an entity handle, **`entmake`** ignores that too.

Return Values

If successful, **`entmake`** returns the entity's list of definition data. If **`entmake`** is unable to create the entity, it returns `nil`.

Completion of a block definition (**`entmake`** of an endblk) returns the block's name rather than the entity data list normally returned.

Examples

The following code creates a red circle (color 62), centered at (4,4) with a radius of 1. The optional layer and linetype fields have been omitted and therefore assume default values.

```
Command: (entmake '((0 . "CIRCLE") (62 . 1) (10 4.0 4.0 0.0) (40 . 1.0)))
((0 . "CIRCLE") (62 . 1) (10 4.0 4.0 0.0) (40 . 1.0))
```

Notes on Using entmake

You cannot create viewport objects with **`entmake`**.

A group 66 code is honored only for insert objects (meaning *attributes follow*). For polyline entities, the group 66 code is forced to a value of 1 (meaning *vertices follow*), and for all other entities it takes a default of 0. The only entity that can follow a polyline entity is a vertex entity.

The group code 2 (block name) of a dimension entity is optional for the **`entmake`** function. If the block name is omitted from the entity definition list, AutoCAD creates a new one. Otherwise, AutoCAD creates the dimension using the name provided.

See Also

The entdel, entget, and entmod functions. In the *Visual LISP Developer's Guide*, refer to "Entity Data Functions" for additional information on creating entities in a drawing, "Adding an Entity to a Drawing" for specifics on using **`entmake`**, and "Creating Complex Entities" for information on creating complex entities.

entmakex

Makes a new object or entity, gives it a handle and entity name (but, does not assign an owner), and then returns the new entity name

`(entmakex` *`[elist]`*`)`

The **`entmakex`** function can define both graphical and nongraphical entities.

Arguments

elist	A list of entity definition data in a format similar to that returned by the **`entget`** function. The *elist* argument must contain all of the information necessary to define the entity. If any required definition data is omitted, **`entmakex`** returns `nil` and the entity is rejected. If you omit optional definition data (such as the layer), **`entmakex`** uses the default value.

Return Values

If successful, **`entmakex`** returns the name of the entity created. If **`entmakex`** is unable to create the entity, the function returns `nil`.

Examples

```
_$ (entmakex '((0 . "CIRCLE") (62 . 1) (10 4.0 3.0 0.0) (40 . 1.0)))
<Entity name: 1d45558>
```

WARNING! Objects and entities without owners are not written out to *.dwg* or *.dxf* files. Be sure to set an owner at some point after using **`entmakex`**. For example, you can use **`dictadd`** to set a dictionary to own an object.

See Also

The entmake function.

entmod

Modifies the definition data of an object (entity)

(entmod *elist*)

The **entmod** function updates database information for the entity name specified by the –1 group in *elist*. The primary mechanism through which AutoLISP updates the database is by retrieving entities with **entget**, modifying the list defining an entity, and updating the entity in the database with **entmod**. The **entmod** function can modify both graphical and nongraphical objects.

Arguments

elist	A list of entity definition data in a format similar to that returned by the **entget** function. For entity fields with floating-point values (such as thickness), **entmod** accepts integer values and converts them to floating point. Similarly, if you supply a floating-point value for an integer entity field (such as color number), **entmod** truncates it and converts it to an integer.

Return Values

If successful, **entmod** returns the *elist* supplied to it. If **entmod** is unable to modify the specified entity, the function returns `nil`.

Examples

The following sequence of commands obtains the properties of an entity, then modifies the entity.

Set the `en1` variable to the name of the first entity in the drawing:

```
Command: (setq en1 (entnext))
<Entity name: 2c90520>
```

Set a variable named `ed` to the entity data of entity `en1`:

```
Command: (setq ed (entget en1))
((-1 . <Entity name: 2c90520>) (0 . "CIRCLE") (5 . "4C") (100 . "AcDbEntity") (67
. 0) (8 . "0") (100 . "AcDbCircle") (10 3.45373 6.21635 0.0) (40 . 2.94827) (210
0.0 0.0 1.0))
```

Changes the layer group in `ed` from layer 0 to layer 1:

Command: **(setq ed (subst (cons 8 "1") (assoc 8 ed) ed))**
((-1 . <Entity name: 2c90520>) (0 . "CIRCLE") (5 . "4C") (100 . "AcDbEntity") (67 . 0) (8 . "1") (100 . "AcDbCircle") (10 3.45373 6.21635 0.0) (40 . 2.94827) (210 0.0 0.0 1.0))

Modify the layer of the `en1` entity in the drawing:

Command: **(entmod ed)**
((-1 . <Entity name: 2c90520>) (0 . "CIRCLE") (5 . "4C") (100 . "AcDbEntity") (67 . 0) (8 . "1") (100 . "AcDbCircle") (10 3.45373 6.21635 0.0) (40 . 2.94827) (210 0.0 0.0 1.0))

Restrictions on Using entmod

There are restrictions on the changes the **`entmod`** function can make:

- An entity's type and handle cannot be changed. If you want to do this, use **`entdel`** to delete the entity, then make a new entity with the **`command`** or **`entmake`** functions.
- The **`entmod`** function cannot change internal fields such as the entity name in the –2 group of a seqend entity—attempts to change such fields are ignored.
- You cannot use the **`entmod`** function to modify a viewport entity.

You can change an entity's space visibility field to 0 or 1 (except for viewport objects). If you use **`entmod`** to modify an entity within a block definition, the modification affects all instances of the block in the drawing.

Before performing an **`entmod`** on vertex entities, you should read or write the polyline entity's header. If the most recently processed polyline entity is different from the one to which the vertex belongs, width information (the 40 and 41 groups) can be lost.

WARNING! You can use **`entmod`** to modify entities within a block definition, but doing so can create a self-referencing block, which will cause AutoCAD to stop.

See Also

The entdel, entget, entmake, and entnext functions. In the *Visual LISP Developer's Guide*, refer to "Modifying an Entity" and "Entity Data Functions and the Graphics Screen."

entnext

Returns the name of the next object (entity) in the drawing

```
(entnext [ename])
```

Arguments

ename	The name of an existing entity.

Return Values

If **`entnext`** is called with no arguments, it returns the entity name of the first nondeleted entity in the database. If an *ename* argument is supplied to **`entnext,`** the function returns the entity name of the first nondeleted entity following *ename* in the database. If there is no next entity in the database, it returns `nil`. The **`entnext`** function returns both main entities and subentities.

Examples

```
(setq e1 (entnext))      ; Sets e1 to the name of the first entity in the drawing
(setq e2 (entnext e1))    ; Sets e2 to the name of the entity following e1
```

Notes

The entities selected by **`ssget`** are main entities, not attributes of blocks or vertices of polylines. You can access the internal structure of these complex entities by walking through the subentities with **`entnext`**. Once you obtain a subentity's name, you can operate on it like any other entity. If you obtain the name of a subentity with **`entnext`**, you can find the parent entity by stepping forward with **`entnext`** until a seqend entity is found, then extracting the –2 group from that entity, which is the main entity's name.

See Also

The entdel, entget, entmake, and entnext functions.

entsel

Prompts the user to select a single object (entity) by specifying a point

`(entsel [msg])`

Arguments

msg	A prompt string to be displayed to users. If omitted, **`entsel`** prompts with the message, "Select object".

Return Values

A list whose first element is the entity name of the chosen object and whose second element is the coordinates (in terms of the current UCS) of the point used to pick the object.

The pick point returned by **`entsel`** does not represent a point that lies on the selected object. The point returned is the location of the crosshairs at the time of selection. The relationship between the pick point and the object will vary depending on the size of the pickbox and the current zoom scale.

Examples

The following AutoCAD command sequence illustrates the use of the **`entsel`** function and the list returned:

```
Command:   line
From point:   1,1
To point:   6,6
To point:   ENTER

Command:   (setq e (entsel "Please choose an object: "))
Please choose an object:   3,3
(<Entity name: 60000014> (3.0 3.0 0.0))
```

Sometimes when operating on objects, you will want to simultaneously select an object and specify the point by which it was selected. Examples of this in AutoCAD can be found in Object Snap and in the BREAK, TRIM, and EXTEND commands. The **`entsel`** function allows AutoLISP programs to perform this operation. It selects a single object, requiring the selection to be a point pick. The current Osnap setting is ignored by this function unless you specifically request it while you are in the function. The **`entsel`** function honors keywords from a preceding call to **`initget`**.

See Also

The entget, entmake, and entnext functions.

entupd

Updates the screen image of an object (entity)

`(entupd` *`ename`*`)`

Arguments

ename	The name of the entity to be updated on the screen.

Return Values

The entity (*ename*) updated, or `nil`, if nothing was updated.

Examples

Assuming that the first entity in the drawing is a 3D polyline with several vertices, the following code modifies and redisplays the polyline:

```
(setq e1 (entnext))          ; Sets e1 to the polyline's entity name
(setq e2 (entnext e1))       ; Sets e2 to its first vertex
(setq ed (entget e2))        ; Sets ed to the vertex data
(setq ed
  (subst '(10 1.0 2.0)
    (assoc 10 ed)            ; Changes the vertex's location in ed
    ed                       ; to point (1,2)
  )
)
(entmod ed)                  ; Moves the vertex in the drawing
(entupd e1)                  ; Regenerates the polyline entity e1
```

Updating Polylines and Blocks

When a 3D (or old-style) polyline vertex or block attribute is modified with **`entmod`**, the entire complex entity is not updated on the screen. The **`entupd`** function can be used to cause a modified polyline or block to be updated on the screen. This function can be called with the entity name of any part of the polyline or block; it need not be the head entity. While **`entupd`** is intended for polylines and blocks with attributes, it can be called for any entity. It always regenerates the entity on the screen, including all subentities.

NOTE If **`entupd`** is used on a nested entity (an entity within a block) or on a block that contains nested entities, some of the entities might not be regenerated. To ensure complete regeneration, you must invoke the REGEN command.

See Also

The entget, entmod, and entnext functions.

eq

Determines whether two expressions are identical

`(eq` ***`expr1 expr2`*****`)`**

The **`eq`** function determines whether *expr1* and *expr2* are bound to the same object (by **`setq`**, for example).

Arguments

expr1	The expression to be compared.
expr2	The expression to compare with *expr1*.

Return Values

`T` if the two expressions are identical, `nil` otherwise.

Examples

Given the following assignments:

```
(setq f1 '(a b c))
(setq f2 '(a b c))
(setq f3 f2)
```

Compare `f1` and `f3`:

```
Command: (eq f1 f3)
nil
```

`eq` returns `nil` because `f1` and `f3`, while containing the same value, do not refer to the same list.

Compare `f3` and `f2`:

Command: **(eq f3 f2)**
T

`eq` returns `T` because `f3` and `f2` refer to the same list.

See Also

The **`= (equal to)`** and equal functions.

equal

Determines whether two expressions are equal

`(equal` ***`expr1 expr2 [fuzz]`*****`)`**

Arguments

expr1	The expression to be compared.
expr2	The expression to compare with *expr1*.
fuzz	A real number defining the maximum amount by which *expr1* and *expr2* can differ and still be considered equal.

When comparing two real numbers (or two lists of real numbers, as in points), the two identical numbers can differ slightly if different methods are used to calculate them. You can specify a *fuzz* amount to compensate for the difference that may result from the different methods of calculation.

Return Values

`T` if the two expressions are equal (evaluate to the same value), `nil` otherwise.

Examples

Given the following assignments:

```
(setq f1 '(a b c))
(setq f2 '(a b c))
(setq f3 f2)
(setq a 1.123456)
(setq b 1.123457)
```

Compare `f1` to `f3`:

Command: **(equal f1 f3)**
T

Compare `f3` to `f2`:

Command: **(equal f3 f2)**
T

Compare `a` to `b`:

Command: **(equal a b)**
nil

The `a` and `b` variables differ by .000001.

Compare `a` to `b`:, with *fuzz* argument of .000001:

Command: **(equal a b 0.000001)**
T

The `a` and `b` variables differ by an amount equal to the specified *fuzz* factor, so **`equal`** considers the variables equal.

Comparing the eq and equal Functions

If the **`eq`** function finds that two lists or atoms are the same, the **`equal`** function also finds them to be the same.

Any *atoms* that the **`equal`** function determines to be the same are also found equivalent by **`eq`**. However, two *lists* that **`equal`** determines to be the same may be found to be different according to the **`eq`** function.

See Also

The **`= (equal to)`** and eq functions.

error

A user-definable error-handling function

`(*error* `*`string`*`)`

If **`*error*`** is not `nil`, it is executed as a function whenever an AutoLISP error condition exists. AutoCAD passes one argument to **`*error*`**, which is a string containing a description of the error.

Your **`*error*`** function can include calls to the **`command`** function without arguments (for example, `(command)`). This will cancel a previous AutoCAD command called with the **`command`** function.

Return Values

This function does not return, except when using vl-exit-with-value.

Examples

The following function does the same thing that the AutoLISP standard error handler does. It prints the word “error,” followed by a description:

```
(defun *error* (msg)
  (princ "error: ")
  (princ msg)
  (princ)
)
```

See Also

The vl-exit-with-error, vl-exit-with-value, vl-catch-all-apply, vl-catch-all-error-message, and vl-catch-all-error-p functions.

eval

Returns the result of evaluating an AutoLISP expression

`(eval `*`expr`*`)`

Arguments

expr	The expression to be evaluated.

Return Values

The result of the expression, after evaluation.

Examples

First, set some variables:

Command: **(setq a 123)**
123

Command: **(setq b 'a)**
A

Now evaluate some expressions:

```
Command: (eval 4.0)
4.0

Command: (eval (abs -10))
10

Command: (eval a)
123

Command: (eval b)
123
```

exit

Forces the current application to quit

```
(exit)
```

If `exit` is called, it returns the error message quit/exit abort and returns to the AutoCAD Command prompt.

See Also

The quit function.

exp

Returns the constant *e* (a real number) raised to a specified power (the natural antilog)

```
(exp num)
```

Arguments

num	A real number.

Return Values

A real (*num*), raised to its natural antilogarithm.

Examples

Command: **(exp 1.0)**
2.71828

Command: **(exp 2.2)**
9.02501

Command: **(exp -0.4)**
0.67032

expand

Allocates additional memory for AutoLISP

```
(expand n-expand)
```

Arguments

n-expand	An integer indicating the amount of additional memory to be allocated. Memory is allocated as follows: ■ *n-alloc free symbols* ■ *n-alloc free strings* ■ *n-alloc free usubrs* ■ *n-alloc free reals* ■ *n-alloc* * *n-expand* cons cells where *n-alloc* is the current segment size.

Return Values

An integer indicating the number of free conses divided by *n-alloc*.

Examples

Set the segment size to 100:

```
_$ (alloc 100)
1000
```

Allocate memory for two additional segments:

```
_$ (expand 2)
82
```

This ensures that AutoLISP now has memory available for at least 200 additional symbols, strings, usubrs and reals each, and 8200 free conses.

See Also

The alloc function.

expt

Returns a number raised to a specified power

```
(expt number power)
```

Arguments

number	Any number.
power	The power to raise *number* to.

Return Values

If both arguments are integers, the result is an integer, otherwise, the result is a real.

Examples

Command: **(expt 2 4)**
16

Command: **(expt 3.0 2.0)**
9.0

fill_image

Draws a filled rectangle in the currently active dialog box image tile

```
(fill_image x1 y1 width height color)
```

The first (upper-left) corner of the rectangle is located at (*x1*,*y1*) and the second (lower-right) corner is located the relative distance (*width*,*height*) from the first corner. The origin (0,0) is the upper-left corner of the image. You can obtain the coordinates of the lower-right corner by calling the dimension functions `dimx_tile` and `dimy_tile`.

The **`fill_image`** function must be used between **`start_image`** and **`end_image`**function calls.

Arguments

x1	*X* coordinate of the upper-left corner of the rectangle located at (*x1*,*y1*). Must be a positive value.
y1	*Y* coordinate of upper-left corner. Must be a positive value.
width	Width of the fill area (in pixels), relative to *x1*.
height	Width of the fill area (in pixels), relative to *y1*.
color	An AutoCAD color number, or one of the logical color numbers shown in the following table:

Symbolic names for color attribute

Color number	ADI mnemonic	Description
–2	BGLCOLOR	Current background of the AutoCAD graphics screen
–15	DBGLCOLOR	Current dialog box background color
–16	DFGLCOLOR	Current dialog box foreground color (text)
–18	LINELCOLOR	Current dialog box line color

Return Values

An integer representing the fill color.

Examples

```
(setq color -2) ;; color of AutoCAD background screen
(fill_image
  0
  0
  (dimx_tile "slide_tile")
  (dimy_tile "slide_tile")
  color
)
(end_image)
```

findfile

Searches the AutoCAD library path for the specified file or directory

`(findfile` ***`filename`*****`)`**

The **`findfile`** function makes no assumption about the file type or extension of *filename*. If *filename* does not specify a drive/directory prefix, **`findfile`** searches the AutoCAD library path. If a drive/directory prefix is supplied, **`findfile`** looks only in that directory.

Arguments

filename	Name of the file or directory to be searched for.

Return Values

A string containing the fully qualified file name, or `nil`, if the specified file or directory is not found.

The file name returned by **`findfile`** is suitable for use with the **`open`** function.

Examples

If the current directory is */AutoCAD 2000i* and it contains the file *abc.lsp*, the following function call retrieves the path name:

```
Command: (findfile "abc.lsp")
"C:\\Program Files\\AutoCAD 2000i\\abc.lsp"
```

If you are editing a drawing in the */AutoCAD 2000i/drawings* directory, the ACAD environment variable is set to */AutoCAD 2000i/support*, and the file *xyz.txt* exists only in the */AutoCAD 2000i/support* directory, then the following command retrieves the path name:

```
Command: (findfile "xyz.txt")
"C:\\Program Files\\AutoCAD 2000i\\support\\xyz.txt"
```

If the file *nosuch* is not present in any of the directories on the library search path, **`findfile`** returns `nil`:

```
Command: (findfile "nosuch")
nil
```

Note that prior to AutoCAD Release 14, **`findfile`** only returned a path if you supplied a valid file name as your argument. If you supplied a directory path, **`findfile`** returned `nil` even if the path existed. For example, the following call to **`findfile`** returns a path name in AutoCAD 2000:

```
Command: (findfile "c:/program files/AutoCAD 2000i")
"C:\\program files\\AutoCAD 2000i"
```

In AutoCAD Release 13, the same command returns `nil`.

fix

Returns the conversion of a real number into the nearest smaller integer

`(fix *number*)`

The **`fix`** function truncates *number* to the nearest integer by discarding the fractional portion.

Arguments

number	A real number.

Return Values

The integer derived from *number*.

If *number* is larger than the largest possible integer (+2,147,483,647 or –2,147,483,648 on a 32-bit platform), **`fix`** returns a truncated real (although integers transferred between AutoLISP and AutoCAD are restricted to 16-bit values).

Examples

```
Command: (fix 3)
3

Command: (fix 3.7)
3
```

float

Returns the conversion of a number into a real number

```
(float number)
```

Arguments

number	Any number.

Return Values

The real number derived from *number*.

Examples

Command: **(float 3)**
3.0

Command: **(float 3.75)**
3.75

foreach

Evaluates expressions for all members of a list

```
(foreach name list [expr...])
```

The **`foreach`** function steps through a list, assigning each element in the list to a variable, and evaluates each expression for every element in the list. Any number of expressions can be specified.

Arguments

name	Variable that each element in the list will be assigned to.
list	List to be stepped through and evaluated.
expr	Expression to be evaluated for each element in *list*.

Return Values

The result of the last *expr* evaluated. If no *expr* is specified, **`foreach`** returns `nil`.

Examples

Print each element in a list:

```
Command: (foreach n '(a b c) (print n))
A
B
C C
```

`foreach` prints each element in the list and returns `C`, the last element. This command is equivalent to the following sequence of commands:

```
(print a)
(print b)
(print c)
```

except that **`foreach`** returns the result of only the last expression evaluated.

function

Tells the Visual LISP compiler to link and optimize an argument as if it were a built-in function

`(function` ***`symbol`*** **`|`** ***`lambda-expr`*****`)`**

The **`function`** function is identical to the **`quote`** function, except it tells the Visual LISP compiler to link and optimize the argument as if it were a built-in function or **`defun`**.

Compiled **`lambda`** expressions that are quoted by **`function`** will contain debugging information when loaded into the Visual LISP IDE.

Arguments

symbol	A symbol naming a function.
lambda-expr	An expression of the following form: `(LAMBDA` *`arguments`* `{`*`S-expression`*`}* )`

Return Values

The result of the evaluated expression.

Examples

The Visual LISP compiler cannot optimize the quoted **`lambda`** expression in the following code:

```
(mapcar
  '(lambda (x) (* x x))
       '(1 2 3))
```

After adding the **`function`** function to the expression, the compiler can optimized the **`lambda`** expression. For example:

```
(mapcar
   (function (lambda (x) (* x x)))
      '(1 2 3))
```

gc

Forces a garbage collection, which frees up unused memory

`(gc)`

See Also

The “Memory Management Functions” topic in the *Visual LISP Developer's Guide*.

gcd

Returns the greatest common denominator of two integers

(gcd *int1 int2*)

Arguments

int1	An integer; must be greater than 0.
int2	An integer; must be greater than 0.

Return Values

An integer representing the greatest common denominator between *int1* and *int2*.

Examples

Command: **(gcd 81 57)**
3

Command: **(gcd 12 20)**
4

get_attr

Retrieves the DCL value of a dialog box attribute

```
(get_attr key attribute)
```

Arguments

key	A string that specifies the tile. This parameter is case-sensitive.
attribute	A string naming the attribute as it appears in the tile's DCL description.

Return Values

A string containing the attribute's initial value as specified in its DCL description.

get_tile

Retrieves the current runtime value of a dialog box tile

```
(get_tile key)
```

Arguments

key	A string that specifies the tile. This parameter is case-sensitive.

Return Values

A string containing the tile's value.

getangle

Pauses for user input of an angle, and returns that angle in radians

```
(getangle [pt] [msg])
```

Arguments

pt	A 2D base point in the current UCS. The *pt* argument, if specified, is assumed to be the first of two points, so the user can show AutoLISP the angle by pointing to one other point. You can supply a 3D base point, but the angle is always measured in the current construction plane.
msg	A string to be displayed to prompt the user.

Return Values

The angle specified by the user, in radians.

The `getangle` function measures angles with the zero-radian direction (set by the ANGBASE system variable) with angles increasing in the counterclockwise direction. The returned angle is expressed in radians with respect to the current construction plane (the *XY* plane of the current UCS, at the current elevation).

Examples

The following code examples show how different arguments can be used with `getangle`:

```
Command: (setq ang (getangle))
Command: (setq ang (getangle '(1.0 3.5)))
Command: (setq ang (getangle "Which way? "))
Command: (setq ang (getangle '(1.0 3.5) "Which way? "))
```

Usage Notes

Users can specify an angle by entering a number in the AutoCAD current angle units format. Although the current angle units format might be in degrees, grads, or some other unit, this function always returns the angle in radians. The user can also show AutoLISP the angle by pointing to two 2D locations on the graphics screen. AutoCAD draws a rubber-band line from the first point to the current crosshairs position to help you visualize the angle.

It is important to understand the difference between the input angle and the angle returned by **`getangle`**. Angles that are passed to **`getangle`** are based on the current settings of ANGDIR and ANGBASE. However, once an angle is provided, it is measured in a counterclockwise direction (ignoring ANGDIR) with zero radians as the current setting of ANGBASE.

The user cannot enter another AutoLISP expression as the response to a **`getangle`** request.

See Also

The illustration and comparison to the getorient function.

getcfg

Retrieves application data from the AppData section of the *acad.cfg* file

`(getcfg` ***`cfgname`*****`)`**

Arguments

cfgname	A string (maximum length of 496 characters) naming the section and parameter value to retrieve.

The *cfgname* argument must be a string of the following form:

`"AppData/`*`application_name/section_name/.../param_name`*`"`

Return Values

Application data, if successful. If *cfgname* is not valid, **`getcfg`** returns `nil`.

Examples

Assuming the WallThk parameter in the AppData/ArchStuff section has a value of 8, the following command retrieves that value:

```
Command: (getcfg "AppData/ArchStuff/WallThk")
"8"
```

See Also

The setcfg function.

getcname

Retrieves the localized or English name of an AutoCAD command

(getcname *cname*)

Arguments

cname	The localized or underscored English command name; must be 64 characters or less in length.

Return Values

If *cname* is not preceded by an underscore (assumed to be the localized command name), **getcname** returns the underscored English command name. If *cname* is preceded by an underscore, **getcname** returns the localized command name. This function returns nil if *cname* is not a valid command name.

Examples

In a French version of AutoCAD, the following is true.

```
(getcname "ETIRER")      returns  "_STRETCH"
(getcname "_STRETCH")    returns  "ETIRER"
```

getcorner

Pauses for user input of a rectangle's second corner

(getcorner *pt [msg]*)

The **getcorner** function takes a base point argument, based on the current UCS, and draws a rectangle from that point as the user moves the crosshairs on the screen.

The user cannot enter another AutoLISP expression in response to a **getcorner** request.

Arguments

pt	A point to be used as the base point.
msg	A string to be displayed to prompt the user.

Return Values

The **`getcorner`** function returns a point in the current UCS, similar to **`getpoint`**. If the user supplies a 3D point, its *Z* coordinate is ignored. The current elevation is used as the *Z* coordinate.

Examples

```
Command: (getcorner '(7.64935 6.02964 0.0))
(17.2066 1.47628 0.0)

Command: (getcorner '(7.64935 6.02964 0.0) "Pick a corner")
Pick a corner(15.9584 2.40119 0.0)
```

getdist

Pauses for user input of a distance

`(getdist [pt] [msg])`

The user can specify the distance by selecting two points, or by specifying just the second point, if a base point is provided. The user can also specify a distance by entering a number in the AutoCAD current distance units format. Although the current distance units format might be in feet and inches (architectural), the **`getdist`** function always returns the distance as a real.

The **`getdist`** function draws a rubber-band line from the first point to the current crosshairs position to help the user visualize the distance.

The user cannot enter another AutoLISP expression in response to a **`getdist`** request.

Arguments

pt	A 2D or 3D point to be used as the base point in the current UCS. If *pt* is provided, the user is prompted for the second point.
msg	A string to be displayed to prompt the user. If no string is supplied, AutoCAD does not display a message.

Return Values

A real number. If a 3D point is provided, the returned value is a 3D distance. However, setting the 64 bit of the **`initget`** function instructs **`getdist`** to ignore the *Z* component of 3D points and to return a 2D distance.

Examples

```
(setq dist (getdist))
(setq dist (getdist '(1.0 3.5)))
(setq dist (getdist "How far "))
(setq dist (getdist '(1.0 3.5) "How far? "))
```

getenv

Returns the string value assigned to a system environment variable

(getenv *variable-name*)

Arguments

variable-name	A string specifying the name of the variable to be read. Environment variable names must be spelled and cased exactly as they are stored in the system registry.

Return Values

A string representing the value assigned to the specified system variable. If the variable does not exist, **getenv** returns nil.

Examples

Assume the system environment variable ACAD is set to */acad/support* and there is no variable named NOSUCH.

```
Command: (getenv "ACAD")
"/acad/support"

Command: (getenv "NOSUCH")
nil
```

Assume that the MaxArray environment variable is set to 10000:

```
Command: (getenv "MaxArray")
"10000"
```

See Also

The setenv function.

getfiled

Prompts the user for a file name with the standard AutoCAD file dialog box, and returns that file name

`(getfiled` *`title default ext flags`*`)`

The **`getfiled`** function displays a dialog box containing a list of available files of a specified extension type. You can use this dialog box to browse through different drives and directories, select an existing file, or specify the name of a new file.

Arguments

title	A string specifying the dialog box label.
default	A default file name to use; can be a null string (`""`).
ext	The default file name extension. If *ext* is passed as a null string (`""`), it defaults to `*` (all file types). If the file type `dwg` is included in the *ext* argument, the **`getfiled`** function displays an image preview in the dialog.
flags	An integer value (a bit-coded field) that controls the behavior of the dialog box. To set more than one condition at a time, add the values together to create a *flags* value between 0 and 15. The following *flags* arguments are recognized by **`getfiled`**: **1** (bit 0) Prompt for the name of a new file to create. Do not set this bit when you prompt for the name of an existing file to open. In the latter case, if the user enters the name of a file that doesn't exist, the dialog box displays an error message at the bottom of the box. If this bit is set and the user chooses a file that already exists, AutoCAD displays an alert box and offers the choice of proceeding with or canceling the operation. **4** (bit 2) Let the user enter an arbitrary file name extension, or no extension at all. If this bit is not set, **`getfiled`** accepts only the extension specified in the *ext* argument and appends this extension

to the file name if the user doesn't enter it in the File text box.

8 (bit 3) If this bit is set and bit 0 is not set, **`getfiled`** performs a library search for the file name entered. If it finds the file and its directory in the library search path, it strips the path and returns only the file name. (It does not strip the path name if it finds that a file of the same name is in a different directory.)

If this bit is not set, **`getfiled`** returns the entire file name, including the path name.

Set this bit if you use the dialog box to open an existing file whose name you want to save in the drawing (or other database).

16 (bit 4) If this bit is set, or if the *default* argument ends with a path delimiter, the argument is interpreted as a path name only. The **`getfiled`** function assumes that there is no default file name. It displays the path in the Look in: line and leaves the File name box blank.

32 (bit 5) If this bit is set and bit 0 is set (indicating that a new file is being specified), users will not be warned if they are about to overwrite an existing file. The alert box to warn users that a file of the same name already exists will not be displayed; the old file will just be replaced.

64 (bit 6) Do not transfer the remote file if the user specifies a URL.

128 (bit 7) Do not allow URLs at all.

Return Values

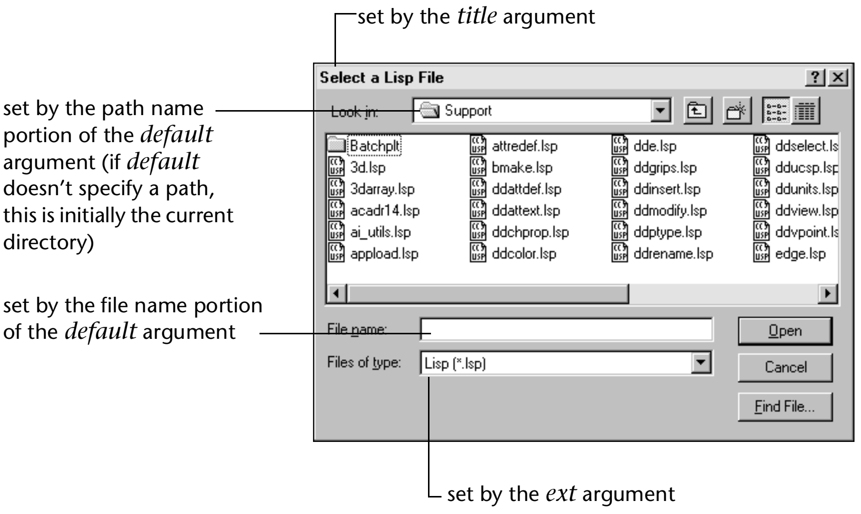
If the dialog box obtains a file name from the user, **`getfiled`** returns a string that specifies the file name; otherwise, it returns `nil`.

Examples

The following call to **`getfiled`** displays the Select a Lisp File dialog box:

```
(getfiled "Select a Lisp File" "c:/program files/autoCAD 2000i/
support/" "lsp" 8)
```

AutoCAD displays the following dialog box as a result:



getint

Pauses for user input of an integer, and returns that integer

`(getint [msg])`

Values passed to **`getint`** can range from –32,768 to +32,767. If the user enters something other than an integer, **`getint`** displays the message "Requires an integer value," and allows the user to try again. The users cannot enter another AutoLISP expression as the response to a **`getint`** request.

Arguments

msg	A string to be displayed to prompt the user; if omitted, no message is displayed.

Return Values

The integer specified by the user; or `nil`, if the user presses ENTER without entering an integer.

Examples

Command: **(setq num (getint))**
15
15

Command: **(setq num (getint "Enter a number:"))**
Enter a number:25
25

Command: **(setq num (getint))**
15.0
Requires an integer value.
15
15

See Also

The initget function in this reference and "The getxxx Functions" in the *Visual LISP Developer's Guide*.

getkword

Pauses for user input of a keyword, and returns that keyword

`(getkword [msg])`

Valid keywords are set prior to the **`getkword`** call with the **`initget`** function. The user cannot enter another AutoLISP expression as the response to a **`getkword`** request.

Arguments

msg	A string to be displayed to prompt the user; if omitted, **`getkword`** does not display a prompting message.

Return Values

A string representing the keyword entered by the user, or `nil`, if the user presses ENTER without typing a keyword. The function also returns `nil` if it was not preceded by a call to **`initget`** to establish one or more keywords.

If the user enters a value that is not a valid keyword, **`getkword`** displays a warning message and prompts the user to try again.

Examples

The following example shows an initial call to **`initget`** that sets up a list of keywords (Yes and No) and disallows null input (*bits* value equal to 1) to the **`getkword`** call that follows:

```
Command: (initget 1 "Yes No")
nil

Command: (setq x (getkword "Are you sure? (Yes or No) "))
Are you sure? (Yes or No) yes
"Yes"
```

The following sequence illustrates what happens if the user enters invalid input in response to **`getkword`**:

```
Command: (initget 1 "Yes No")
nil

Command: (setq x (getkword "Are you sure? (Yes or No) "))
Are you sure? (Yes or No) Maybe
Invalid option keyword.
Are you sure? (Yes or No) yes
"Yes"
```

The user's response was not one of the keywords defined by the preceding **`initget`**, so **`getkword`** issued an error message and then prompted the user again with the string supplied in the *msg* argument.

See Also

The initget function in this reference and"The getxxx Functions" in the *Visual LISP Developer's Guide*.

getorient

Pauses for user input of an angle, and returns that angle in radians

`(getorient [pt] [msg])`

The **`getorient`** function measures angles with the zero-radian direction to the right (east) and angles that are increasing in the counterclockwise direction. The angle input by the user is based on the current settings of ANGDIR and ANGBASE, but once an angle is provided, it is measured in a counterclockwise direction, with zero radians being to the right (ignoring ANGDIR and

ANGBASE). Therefore, some conversion must take place if you select a different zero-degree base or a different direction for increasing angles by using the UNITS command or the ANGBASE and ANGDIR system variables.

Use **`getangle`** when you need a rotation amount (a relative angle). Use **`getorient`** to obtain an orientation (an absolute angle).

The user cannot enter another AutoLISP expression as the response to a **`getorient`** request.

Arguments

pt	A 2D base point in the current UCS.
	The *pt* argument, if specified, is assumed to be the first of two points, so that the user can show AutoLISP the angle by pointing to one other point. You can supply a 3D base point, but the angle is always measured in the current construction plane.
msg	A string to be displayed to prompt the user.

Return Values

The angle specified by the user, in radians, with respect to the current construction plane.

Examples

```
Command: (setq pt1 (getpoint "Pick point: "))
(4.55028 5.84722 0.0)

Command: (getorient pt1 "Pick point: ")
5.61582
```

See Also

The getangle function in this reference and "The getxxx Functions" in the *Visual LISP Developer's Guide*.

getpoint

Pauses for user input of a point, and returns that point

(getpoint *[pt] [msg]*)

The user can specify a point by pointing or by entering a coordinate in the current units format. If the *pt* argument is present, AutoCAD draws a rubber-band line from that point to the current crosshairs position.

The user cannot enter another AutoLISP expression in response to a **getpoint** request.

Arguments

pt	A 2D or 3D base point in the current UCS. Note that **getpoint** will accept a single integer or real number as the *pt* argument, and use the AutoCAD direct distance entry mechanism to determine a point. This mechanism uses the value of the LASTPOINT system variable as the starting point, the *pt* input as the distance, and the current cursor location as the direction from LASTPOINT. The result is a point that is the specified number of units away from LASTPOINT in the direction of the current cursor location.
msg	A string to be displayed to prompt the user.

Return Values

A 3D point, expressed in terms of the current UCS.

Examples

```
(setq p (getpoint))
(setq p (getpoint "Where? "))
(setq p (getpoint '(1.5 2.0) "Second point: "))
```

See Also

The getcorner and initget functions in this reference and “The getxxx Functions” in the *Visual LISP Developer’s Guide*.

getreal

Pauses for user input of a real number, and returns that real number

`(getreal [msg])`

The user cannot enter another AutoLISP expression as the response to a **`getreal`** request.

Arguments

msg	A string to be displayed to prompt the user.

Return Values

The real number entered by the user.

Examples

```
(setq val (getreal))
(setq val (getreal "Scale factor: "))
```

getstring

Pauses for user input of a string, and returns that string

`(getstring [cr] [msg])`

The user cannot enter another AutoLISP expression as the response to a **`getstring`** request.

Arguments

cr	If supplied and not `nil`, this argument indicates that users can include blanks in their input string (and must terminate the string by pressing ENTER). Otherwise, the input string is terminated by space or ENTER.
msg	A string to be displayed to prompt the user.

Return Values

The string entered by the user, or `nil`, if the user pressed ENTER without typing a string.

If the string is longer than 132 characters, **`getstring`** returns only the first 132 characters of the string. If the input string contains the backslash character (\), **`getstring`** converts it to two backslash characters (\\). This allows you to use returned values containing file name paths in other functions.

Examples

Command: **(setq s (getstring "What's your first name? "))**
What's your first name? **Gary**
"Gary"

Command: **(setq s (getstring T "What's your full name? "))**
What's your full name? **Gary Indiana Jones**
"Gary Indiana Jones"

Command: **(setq s (getstring T "Enter filename: "))**
Enter filename: **c:\my documents\vlisp\secrets**
"c:\\my documents\\vlisp\\secrets"

See Also

The initget function.

getvar

Retrieves the value of an AutoCAD system variable

```
(getvar varname)
```

Arguments

varname	A string or symbol that names a system variable. See the *Command Reference* for a list of current AutoCAD system variables.

Return Values

The value of the system variable, or `nil`, if *varname* is not a valid system variable.

Examples

Get the current value of the fillet radius:

```
Command: (getvar 'FILLETRAD)
0.25
```

See Also

The setvar function.

graphscr

Displays the AutoCAD graphics screen

`(graphscr)`

This function is equivalent to the GRAPHSCR command or pressing the Flip Screen function key. The **`textscr`** function is the complement of **`graphscr`**.

Returns

`nil`

See Also

The textscr function.

grclear

Clears the current viewport (obsolete function)

`(grclear)`

Returns

`nil`

grdraw

Draws a vector between two points, in the current viewport

(grdraw *from to color [highlight]*)

Arguments

from	2D or 3D points (lists of two or three reals) specifying one endpoint of the vector in terms of the current UCS. AutoCAD clips the vector to fit the screen.
to	2D or 3D points (lists of two or three reals) specifying the other endpoint of the vector in terms of the current UCS. AutoCAD clips the vector to fit the screen.
color	An integer identifying the color used to draw the vector. A –1 signifies *XOR ink*, which complements anything it draws over and which erases itself when overdrawn.
highlight	An integer, other than zero, indicating that the vector is to be drawn using the default highlighting method of the display device (usually dashed). If *highlight* is omitted or is zero, **`grdraw`** uses the normal display mode.

Return Values

`nil`

See Also

The grvecs function for a routine that draws multiple vectors.

grread

Reads values from any of the AutoCAD input devices

(grread *[track] [allkeys [curtype]]*)

Only specialized AutoLISP routines need this function. Most input to AutoLISP should be obtained through the various **get*xxx*** functions.

Arguments

track	If supplied and not `nil`, this argument enables the return of coordinates from a pointing device as it is moved
allkeys	An integer representing a code that tells **grread** what functions to perform. The *allkeys* bit code values can be added together for combined functionality. The following values can be specified: **1** (bit 0) Return *drag mode* coordinates. If this bit is set and the user moves the pointing device instead of selecting a button or pressing a key, **grread** returns a list where the first member is a type 5 and the second member is the *(X,Y)* coordinates of the current pointing device (mouse or digitizer) location. This is how AutoCAD implements dragging. **2** (bit 1) Return all key values, including function and cursor key codes, and don't move the cursor when the user presses a cursor key. **4** (bit 2) Use the value passed in the *curtype* argument to control the cursor display. **8** (bit 3) Don't display the error: console break message when the user presses ESC .
curtype	An integer indicating the type of cursor to be displayed. The *allkeys* value for bit 2 must be set for the *curtype* values to take effect. The *curtype* argument affects only the cursor type during the current **grread** function call. You can specify one of the following values for *curtype*: **0** Display the normal crosshairs. **1** Do not display a cursor (no crosshairs). **2** Display the object-selection "target" cursor.

Return Values

The **`grread`** function returns a list whose first element is a code specifying the type of input. The second element of the list is either an integer or a point, depending on the type of input. The return values are listed in the following table:

grread return values

First element		Second element	
Value	**Type of input**	**Value**	**Description**
2	Keyboard input	varies	Character code
3	Selected point	3D point	Point coordinates
4	Screen/pull-down menu item (from pointing device)	0 to 999 1001 to 1999 2001 to 2999 3001 to 3999 … and so on, to 16001 to 16999	Screen menu box no. POP1 menu box no. POP2 menu box no. POP3 menu box no. ... and so on, to POP16 menu box no.
5	Pointing device (returned only if tracking is enabled)	3D point	Drag mode coordinate
6	BUTTONS menu item	0 to 999 1000 to 1999 2000 to 2999 3000 to 3999	BUTTONS1 menu button no. BUTTONS2 menu button no. BUTTONS3 menu button no. BUTTONS4 menu button no.
7	TABLET1 menu item	0 to 32767	Digitized box no.
8	TABLET2 menu item	0 to 32767	Digitized box no.
9	TABLET3 menu item	0 to 32767	Digitized box no.
10	TABLET4 menu item	0 to 32767	Digitized box no.
11	AUX menu item	0 to 999 1000 to 1999 2000 to 2999 3000 to 3999	AUX1 menu button no. AUX2 menu button no. AUX3 menu button no. AUX4 menu button no.
12	Pointer button (follows a type 6 or type 11 return)	3D point	Point coordinates

Handling User Input with grread

Entering ESC while a **grread** is active aborts the AutoLISP program with a keyboard break (unless the *allkeys* argument has disallowed this). Any other input is passed directly to **grread**, giving the application complete control over the input devices.

If the user presses the pointer button within a screen menu or pull-down menu box, **grread** returns a type 6 or type 11 code, but in a subsequent call, it does not return a type 12 code: the type 12 code follows type 6 or type 11 only when the pointer button is pressed while it is in the graphics area of the screen.

It is important to clear the code 12 data from the buffer before attempting another operation with a pointer button or an auxiliary button. To accomplish this, perform a nested **grread** like this:

```
(setq code_12 (grread (setq code (grread))))
```

This sequence captures the value of the code 12 list as streaming input from the device.

grtext

Writes text to the status line or to screen menu areas

(grtext *[box text [highlight]]*)

This function displays the supplied text in the menu area; it does not change the underlying menu item. The **grtext** function can be called with no arguments to restore all text areas to their standard values.

Arguments

box	An integer specifying the location in which to write the text.
text	A string that specifies the text to be written to the screen menu or status line location. The *text* argument is truncated if it is too long to fit in the available area.
highlight	An integer that selects or deselects a screen menu location.

If called without arguments, **grtext** restores all text areas to their standard values. If called with only one argument, **grtext** results in an error.

Return Values

The string passed in the *text* argument, if successful, and `nil`, if unsuccessful or no arguments are supplied.

Screen Menu Area

Setting *box* to a positive or zero value specifies a screen menu location. Valid *box* values range from 0 to the highest-numbered screen menu box minus 1. The SCREENBOXES system variable reports the maximum number of screen menu boxes. If the *highlight* argument is supplied as a positive integer, **`grtext`** highlights the text in the designated box. Highlighting a box automatically dehighlights any other box already highlighted. If *highlight* is zero, the menu item is dehighlighted. If *highlight* is a negative number, it is ignored. On some platforms, the text must first be written without the *highlight* argument and then must be highlighted. Highlighting of a screen menu location works only when the cursor is not in that area.

Status Line Area

If **`grtext`** is called with a *box* value of –1, it writes the text into the mode status line area. The length of the mode status line differs from display to display (most allow at least 40 characters). The following code uses the **`$(linelen)`** DIESEL expression to report the length of the mode status area.

```
(setq modelen (menucmd "M=$(linelen)"))
```

If a *box* value of –2 is used, **`grtext`** writes the text into the coordinate status line area. If coordinate tracking is turned on, values written into this field are overwritten as soon as the pointer sends another set of coordinates. For both –1 or –2 *box* values, the *highlight* argument is ignored.

grvecs

Draws multiple vectors on the graphics screen

`(grvecs` ***`vlist [trans]`*** **`)`**

Arguments

vlist	A vector list is comprised of a series of optional color integers and two point lists. See Vector List Format for details on how to format *vlist*.

trans	A transformation matrix used to change the location or proportion of the vectors defined in your vector list. This matrix is a list of four lists of four real numbers.

Return Values

```
nil
```

Vector List Format

The format for *vlist* is as follows:

```
([color1] from1 to1 [color2] from2 to2 ...)
```

The color value applies to all succeeding vectors until *vlist* specifies another color. AutoCAD colors are in the range 0–255. If the color value is greater than 255, succeeding vectors are drawn in *XOR ink*, which complements anything it draws over and which erases itself when overdrawn. If the color value is less than zero, the vector is highlighted. Highlighting depends on the display device. Most display devices indicate highlighting by a dashed line, but some indicate it by using a distinctive color.

A pair of point lists, *from* and *to*, specify the endpoints of the vectors, expressed in the current UCS. These can be 2D or 3D points. You must pass these points as pairs—two successive point lists—or the **grvecs** call will fail.

AutoCAD clips the vectors as required to fit on the screen.

Examples

The following code draws five vertical lines on the graphics screen, each a different color:

```
(grvecs '(1  (1 2)(1 5)        Draws a red line from (1,2) to (1,5)
          2  (2 2)(2 5)        Draws a yellow line from (2,2) to (2,5)
          3  (3 2)(3 5)        Draws a green line from (3,2) to (3,5)
          4  (4 2)(4 5)        Draws a cyan line from (4,2) to (4,5)
          5  (5 2)(5 5)        Draws a blue line from (5,2) to (5,5)
) )
```

The following matrix represents a uniform scale of 1.0 and a translation of 5.0,5.0,0.0. If this matrix is applied to the preceding list of vectors, they will be offset by 5.0,5.0,0.0.

```
'( (1.0 0.0 0.0 5.0)
   (0.0 1.0 0.0 5.0)
   (0.0 0.0 1.0 0.0)
   (0.0 0.0 0.0 1.0)
)
```

See Also

The nentselp function for more information on transformation matrixes and the grdraw function for a routine that draws a vector between two points.

handent

Returns an object (entity) name based on its handle

```
(handent handle)
```

The **handent** function returns the entity name of both graphic and non-graphic entities.

Arguments

handle	A string identifying an entity handle.

Return Values

If successful, **handent** returns the entity name associated with *handle* in the current editing session. If **handent** is passed an invalid handle or a handle not used by any entity in the current drawing, it returns `nil`.

The **handent** function returns entities that have been deleted during the current editing session. You can undelete them with the **entdel** function.

An entity's name can change from one editing session to the next, but an entity's handle remains constant.

Examples

```
Command: (handent "5A2")
<Entity name: 60004722>
```

Used with the same drawing but in another editing session, the same call might return a different entity name. Once obtained, you can use the entity name to manipulate the entity with any of the entity-related functions.

help

Invokes the help facility

```
(help [helpfile [topic [command]]])
```

Arguments

helpfile	A string naming the help file. The file extension is not required with the *helpfile* argument. If a file extension is provided, AutoCAD looks only for a file with the exact name specified. If no file extension is provided, AutoCAD looks for *helpfile* with an extension of *.chm*. If no file of that name is found, AutoCAD looks for a file with an extension of *.hlp*.
topic	A string identifying a Help topic ID. If you are calling a topic within a CHM file, provide the file name without the extension; AutoCAD adds an *.htm* extension.
command	A string that specifies the initial state of the Help window. The *command* argument is a string used by the uCommand (in HTML Help) or the fuCommand (in WinHelp) argument of the HtmlHelp() and WinHelp() functions as defined in the Microsoft Windows SDK. For HTML Help files, the *command* parameter can be HH_ALINK_LOOKUP or HH_DISPLAY_TOPIC. For Windows Help files, the *command* parameter can be HELP_CONTENTS, HELP_HELPONHELP, or HELP_PARTIALKEY.

Return Values

The *helpfile* string, if successful, otherwise `nil`. If you use **`help`** without any arguments, it returns an empty string (`""`) if successful, and `nil` if it fails.

The only error condition that the **`help`** function returns to the application is the existence of the file specified by *helpfile*. All other error conditions are reported to the user through a dialog box.

Examples

The following code calls **`help`** to display the information on `MYCOMMAND` in the help file *achelp.chm*:

```
(help "achelp.chm" "mycommand")
```

See Also

The setfunhelp function associates context-sensitive help (when the user presses `F1`) with a user-defined command.

if

Conditionally evaluates expressions

`(if testexpr thenexpr [elseexpr])`

Arguments

testexpr	Expression to be tested.
thenexpr	Expression evaluated if *testexpr* is not `nil`.
elseexpr	Expression evaluated if *testexpr* is `nil`.

Return Values

The **`if`** function returns the value of the selected expression. If *elseexpr* is missing and *testexpr* is `nil`, then **`if`** returns `nil`.

Examples

Command: **(if (= 1 3) "YES!!" "no.")**
"no."

Command: **(if (= 2 (+ 1 1)) "YES!!")**
"YES!!"

Command: **(if (= 2 (+ 3 4)) "YES!!")**
nil

See Also

The progn function.

initdia

Forces the display of the next command's dialog box

`(initdia` *`[dialogflag]`*`)`

Currently, the following commands make use of the **`initdia`** function: ATTDEF, ATTEXT, BHATCH, BLOCK, COLOR, IMAGE, IMAGEADJUST, INSERT, LAYER, LINETYPE, MTEXT, PLOT, RENAME, STYLE, TOOLBAR, and VIEW.

Arguments

dialogflag	An integer. If this argument is not present or is present and nonzero, the next use (and next use only) of a command will display that command's dialog box rather than its command line prompts. If *dialogflag* is zero, any previous call to this function is cleared, restoring the default behavior of presenting the command line interface.

Return Values

`nil`

Examples

Issue the PLOT command without calling **`initdia`** first:

Command: **(command "_.PLOT")**
plot
Enter a layout name <Model>: nil
Enter a layout name <Model>:

AutoCAD prompts for user input in the command window.

Use the following sequence of function calls to make AutoCAD display the Plot dialog box:

```
(initdia)
(command "_.PLOT")
```

initget

Establishes keywords for use by the next user-input function call

(initget *[bits] [string]*)

The functions that honor keywords are **getint**, **getreal**, **getdist**, **getangle**, **getorient**, **getpoint**, **getcorner**, **getkword**, **entsel**, **nentsel**, and **nentselp**. The **getstring** function is the only user-input function that does not honor keywords.

The keywords are checked by the next user-input function call when the user does not enter the expected type of input (for example, a point to **getpoint**). If the user input matches a keyword from the list, the function returns that keyword as a string result. The application can test for the keywords and perform the action associated with each one. If the user input is not an expected type and does not match a keyword, AutoCAD asks the user to try again. The **initget** bit values and keywords apply only to the next user-input function call.

If **initget** sets a control bit and the application calls a user-input function for which the bit has no meaning, the bit is ignored.

If the user input fails one or more of the specified conditions (as in a zero value when zero values are not allowed), AutoCAD displays a message and asks the user to try again.

Arguments

bits	A bit-coded integer that allows or disallows certain types of user input. The bits can be added together in any combination to form a value between 0 and 255. If no *bits* argument is supplied, zero (no conditions) is assumed. The bit values are as follows: **1** (bit 0) Prevents the user from responding to the request by entering only ENTER. **2** (bit 1) Prevents the user from responding to the request by entering zero.

4 (bit 2) Prevents the user from responding to the request by entering a negative value.

8 (bit 3) Allows the user to enter a point outside the current drawing limits. This condition applies to the next user-input function even if the AutoCAD system variable LIMCHECK is currently set.

16 (bit 4) (Not currently used.)

32 (bit 5) Uses dashed lines when drawing a rubber-band line or box. For those functions with which the user can specify a point by selecting a location on the graphics screen, this bit value causes the rubber-band line or box to be dashed instead of solid. (Some display drivers use a distinctive color instead of dashed lines.) If the system variable POPUPS is 0, AutoCAD ignores this bit.

64 (bit 6) Prohibits input of a *Z* coordinate to the `getdist` function; lets an application ensure that this function returns a 2D distance.

128 (bit 7) Allows arbitrary input as if it is a keyword, first honoring any other control bits and listed keywords. This bit takes precedence over bit 0; if bits 7 and 0 are set and the user presses ENTER, a null string is returned.

NOTE Future versions of AutoLISP may use additional **`initget`** control bits, so avoid setting bits that are not listed here.

string A string representing a series of keywords. See “Keyword Specifications” on page 107 for information on defining keywords.

Return Values

`nil`

Function Applicable Control Bits

The special control values are honored only by those ***getxxx*** functions for which they make sense, as indicated in the following table:

User-input functions and applicable control bits

Function	Honors key words	Control bits values						
		No null (1)	No zero (2)	No negative (4)	No limits (8)	Uses dashes (32)	2D distance (64)	Arbitrary Input (128)
`getint`	•	•	•	•				•
`getreal`	•	•	•	•				•
`getdist`	•	•	•	•		•	•	•
`getangle`	•	•	•			•		•
`getorient`	•	•	•			•		•
`getpoint`	•	•			•	•		•
`getcorner`	•	•			•	•		•
`getkword`	•	•						•
`entsel`	•							
`nentsel`	•							
`nentselp`	•							

Keyword Specifications

The *string* argument is interpreted according to the following rules:

1 Each keyword is separated from the following keyword by one or more spaces. For example, `"Width Height Depth"` defines three keywords.

2 Each keyword can contain only letters, numbers, and hyphens (–).

There are two methods for abbreviating keywords:

- The required portion of the keyword is specified in uppercase characters, and the remainder of the keyword is specified in lowercase characters. The uppercase abbreviation can be anywhere in the keyword (for example, `"LType"`, `"eXit"`, or `"toP"`).
- The *entire* keyword is specified in uppercase characters, and it is followed immediately by a comma, which is followed by the required characters (for example, `"LTYPE,LT"`). The keyword characters in this case must include the first letter of the keyword, which means that `"EXIT,X"` is not valid.

The two brief examples, `"LType"` and `"LTYPE,LT"`, are equivalent: if the user types **LT** (in either uppercase or lowercase letters), this is sufficient to identify the keyword. The user can enter characters that *follow* the required portion of the keyword, provided they don't conflict with the specification. In the example, the user could also enter **LTY** or **LTYP**, but **L** would not be sufficient.

If *string* shows the keyword entirely in uppercase *or* lowercase characters with no comma followed by a required part, AutoCAD recognizes the keyword only if the user enters all of it.

The **`initget`** function provides support for localized keywords. The following syntax for the keyword string allows input of the localized keyword while it returns the language independent keyword:

```
"local1 local2 localn _indep1 indep2 indepn"
```

where *local1* through *localn* are the localized keywords, and *indep1* through *indepn* are the language-independent keywords.

There must always be the same number of localized keywords as language-independent keywords, and the first language-independent keyword is prefixed by an underscore as shown in the following example:

```
(initget "Abc Def _Ghi Jkl")
(getkword "\nEnter an option (Abc/Def): ")
```

Entering **A** returns Ghi and entering **_J** returns Jkl.

See Also

The "Control of User-Input Function Conditions" topic in the *Visual LISP Developer's Guide*.

inters

Finds the intersection of two lines

`(inters` *`pt1 pt2 pt3 pt4 [onseg]`*`)`

All points are expressed in terms of the current UCS. If all four point arguments are 3D, **`inters`** checks for 3D intersection. If any of the points are 2D, **`inters`** projects the lines onto the current construction plane and checks only for 2D intersection.

Arguments

pt1	One endpoint of the first line.
pt2	The other endpoint of the first line.
pt3	One endpoint of the second line.
pt4	The other endpoint of the second line.
onseg	If specified as `nil`, the lines defined by the four *pt* arguments are considered infinite in length. If the *onseg* argument is omitted or is not `nil`, the intersection point must lie on both lines or **`inters`** returns `nil`.

Return Values

If the *onseg* argument is present and is `nil`, **`inters`** returns the point where the lines intersect, even if that point is off the end of one or both of the lines. If the *onseg* argument is omitted or is not `nil`, the intersection point must lie on both lines or **`inters`** returns `nil`. The **`inters`** function returns `nil` if the two lines do not intersect.

Examples

```
(setq a '(1.0 1.0) b '(9.0 9.0))
(setq c '(4.0 1.0) d '(4.0 2.0))
```

Command: **(inters a b c d)**
nil

Command: **(inters a b c d T)**
nil

Command: **(inters a b c d nil)**
(4.0 4.0)

itoa

Returns the conversion of an integer into a string

`(itoa` *`int`*`)`

Arguments

int	An integer.

Return Values

A string derived from int.

Examples

Command: **(itoa 33)**
"33"

Command: **(itoa -17)**
"-17"

See Also

The atoi function.

lambda

Defines an anonymous function

`(lambda` *`arguments expr`*`...)`

Use the **`lambda`** function when the overhead of defining a new function is not justified. It also makes the programmer's intention more apparent by laying out the function at the spot where it is to be used. This function returns the value of its last *expr*, and is often used in conjunction with **`apply`** and/or **`mapcar`** to perform a function on a list.

Arguments

arguments	Arguments passed to an expression.
expr	An AutoLISP expression.

Return Values

The value of the last *expr*.

Examples

The following examples demonstrate the **lambda** function from the Visual LISP Console window:

```
_$ (apply '(lambda (x y z)
          (* x (- y z))
        )
        '(5 20 14)
)
30

_$ (setq counter 0)
(mapcar '(lambda (x)
          (setq counter (1+ counter))
          (* x 5)
        )
        '(2 4 -6 10.2)
)
0
(10 20 -30 51.0)
```

last

Returns the last element in a list

```
(last lst)
```

Arguments

lst	A list.

Return Values

An atom or a list.

Examples

Command: **(last '(a b c d e))**
E

Command: **(last '(a b c (d e)))**
(D E)

layoutlist

Returns a list of all paper space layouts in the current drawing

```
(layoutlist)
```

Return Values

A list of strings.

Examples

Command: **(layoutlist)**
("Layout1" "Layout2")

length

Returns an integer indicating the number of elements in a list

```
(length lst)
```

Arguments

lst	A list.

Return Values

An integer.

Examples

Command: **(length '(a b c d))**
4

Command: **(length '(a b (c d)))**
3

Command: **(length '())**
0

See Also

The vl-list-length function.

list

Takes any number of expressions, and combines them into one list

```
(list [expr...])
```

This function is frequently used to define a 2D or 3D point variable (a list of two or three reals).

Arguments

expr	An AutoLISP expression.

Return Values

A list, unless no expressions are supplied, in which case **`list`** returns `nil`.

Examples

```
_$ (list 'a 'b 'c)
(A B C)
```

```
_$ (list 'a '(b c) 'd)
(A (B C) D)
```

```
_$ (list 3.9 6.7)
(3.9 6.7)
```

As an alternative to using the **`list`** function, you can explicitly quote a list with the **`quote`** function if there are no variables or undefined items in the list. The single quote character (`'`) is defined as the quote function.

```
_$ '(3.9 6.7)                 means the same as   (list 3.9 6.7)
```

This can be useful for creating association lists and defining points.

See Also

The quote, vl-list*, and vl-list-length functions.

listp

Verifies that an item is a list

```
(listp item)
```

Arguments

item	Any atom, list, or expression.

Return Values

`T` if *item* is a list, `nil` otherwise. Because `nil` is both an atom and a list, the **`listp`** function returns `T` when passed `nil`.

Examples

Command: **(listp '(a b c))**
T

Command: **(listp 'a)**
nil

Command: **(listp 4.343)**
nil

Command: **(listp nil)**
T

Command: **(listp (setq v1 '(1 2 43)))**
T

See Also

The vl-list* and vl-list-length functions.

load

Evaluates the AutoLISP expressions in a file

`(load` ***`filename [onfailure]`*** **`)`**

The **`load`** function can be used from within another AutoLISP function, or even recursively (in the file being loaded).

Arguments

filename	A string that represents the file name. If the *filename* argument does not specify a file extension, **`load`** adds an extension to the name when searching for a file to load. The function will try several extensions, if necessary, in the following order: ■ *.vlx* ■ *.fas* ■ *.lsp*

	As soon as `load` finds a match, it stops searching and loads the file.
	The *filename* can include a directory prefix, as in "c:/ *function/test1*". A forward slash (/) or two backslashes (\\) are valid directory delimiters. If you don't include a directory prefix in the *filename* string, `load` searches the AutoCAD library path for the specified file. If the file is found anywhere on this path, `load` then loads the file.
onfailure	A value returned if `load` fails.
	If the *onfailure* argument is a valid AutoLISP function, it is evaluated. In most cases, the *onfailure* argument should be a string or an atom. This allows an AutoLISP application calling `load` to take alternative action upon failure.

Return Values

Unspecified, if successful. If `load` fails, it returns the value of *onfailure*; if *onfailure* is not defined, failure results in an error message.

Examples

For the following examples, assume that file */fred/test1.lsp* contains the expressions

```
(defun MY-FUNC1 (x)
            ...function body...
)
(defun MY-FUNC2 (x)
            ...function body...
```

and that no file named *test2* with a *.lsp*, *.fas*, or *.vlx* extension exists:

Command: **(load** "**/fred/test1**"**)**
MY-FUNC2

Command: **(load** "**\\fred\\test1**"**)**
MY-FUNC2

Command: **(load** "**/fred/test1**" "**bad**"**)**
MY-FUNC2

Command: **(load** "**test2**" "**bad**"**)**
"bad"

Command: **(load** "**test2**"**)** causes an AutoLISP error

See Also

The defun and vl-load-all functions in this reference, and "Symbol and Function Handling" in the *Visual LISP Developer's Guide*.

load_dialog

Loads a DCL file

`(load_dialog dclfile)`

The **`load_dialog`** function searches for files according to the AutoCAD library search path.

This function is the complement of **`unload_dialog`**. An application can load multiple DCL files with multiple **`load_dialog`** calls.

Arguments

dclfile	A string that specifies the DCL file to load. If the *dclfile* argument does not specify a file extension, *.dcl* is assumed.

Return Values

A positive integer value (`dcl_id`) if successful, or a negative integer if **`load_dialog`** can't open the file. The `dcl_id` is used as a handle in subsequent **`new_dialog`** and **`unload_dialog`** calls.

log

Returns the natural log of a number as a real number

`(log num)`

Arguments

num	A positive number.

Return Values

A real number.

Examples

Command: **(log 4.5)**
1.50408

Command: **(log 1.22)**
0.198851

logand

Returns the result of the logical bitwise AND of a list of integers

```
(logand [int int...])
```

Arguments

int	An integer.

Return Values

An integer (0, if no arguments are supplied).

Examples

Command: **(logand 7 15 3)**
3

Command: **(logand 2 3 15)**
2

Command: **(logand 8 3 4)**
0

logior

Returns the result of the logical bitwise inclusive OR of a list of integers

```
(logior [int int...])
```

Arguments

int	An integer.

Return Values

An integer (0, if no arguments are supplied).

Examples

Command: **(logior 1 2 4)**
7

Command: **(logior 9 3)**
11

lsh

Returns the logical bitwise shift of an integer by a specified number of bits

```
(lsh [int numbits])
```

Arguments

int	An integer.
numbits	Number of bits to shift *int*. If *numbits* is positive, *int* is shifted to the left; if *numbits* is negative, *int* is shifted to the right. In either case, zero bits are shifted in, and the bits shifted out are discarded. If *numbits* is not specified, no shift occurs.

Return Values

The value of *int* after the bitwise shift. The returned value is positive if the significant bit (bit number 31) contains a 0 after the shift operation, otherwise it is negative. If no arguments are supplied, `lsh` returns 0.

The behavior is different from other languages (>> & << of C, C++, or Java) where more than 32 left shifts (of a 32 bit integer) results 0. In right shift also the integer appears again on every 32 shifts.

Examples

Command: **(lsh 2 1)**
4

Command: **(lsh 2 -1)**
1

Command: **(lsh 40 2)**
160

mapcar

Returns a list of the result of executing a function with the individual elements of a list or lists are supplied as arguments to the function

```
(mapcar function list1... listn)
```

Arguments

function	A function.
list1... listn	One or more lists. The number of lists must match the number of arguments required by *function*.

Return Values

A list.

Examples

Command: **(setq a 10 b 20 c 30)**
30

Command: **(mapcar '1+ (list a b c))**
(11 21 31)

This is equivalent to the following series of expressions:

```
(1+ a)
(1+ b)
(1+ c)
```

except that **mapcar** returns a list of the results.

The **lambda** function can specify an anonymous function to be performed by **mapcar**. This is useful when some of the function arguments are constant or are supplied by some other means. The following example, entered from the Visual LISP Console window, demonstrates the use of **lambda** with **mapcar**:

```
_$ (mapcar  '(lambda (x)
          (+ x 3)
          )
         '(10 20 30)
)
(13 23 33)
```

max

Returns the largest of the numbers given

(max [*number number*...])

Arguments

number	A number.

Return Values

A number. If any of the arguments are real numbers, a real is returned, otherwise an integer is returned. If no argument is supplied, **max** returns 0.

Examples

Command: **(max 4.07 -144)**
4.07

Command: **(max -88 19 5 2)**
19

Command: **(max 2.1 4 8)**
8.0

mem

Displays the current state of the AutoLISP memory

```
(mem)
```

The **`mem`** function displays statistics on AutoLISP memory usage. The first line of this statistics report contains the following information:

GC calls	Number of garbage collection calls since AutoLISP started.
GC run time	Total time spent collecting garbage (in milliseconds).

LISP objects are allocated in dynamic (heap) memory that is organized in segments and divided into pages. Memory is described under the heading, "Dynamic memory segments statistics:"

PgSz	Dynamic memory page size (in KB).
Used	Number of pages used.
Free	Number of free (empty) pages.
FMCL	Largest contiguous area of free pages.
Segs	Number of segments allocated.
Type	Internal description of the types of objects allocated in this segment. These include:

lisp stacks—LISP internal stacks

bytecode area—compiled code function modules

CONS memory—CONS objects

::new—untyped memory requests served using this segment

DM Str—dynamic string bodies

DMxx memory—all other LISP nodes

bstack body—internal structure used for IO operations

The final line in the report lists the minimal segment size and the number of allocated segments. AutoLISP keeps a list of no more than three free segments, in order to save system calls for memory requests.

All heap memory is global; that is, all AutoCAD documents share the same heap. This could change in future releases of AutoCAD.

Note that **`mem`** does not list all memory requested from the operating system, only those requests served by the AutoLISP Dynamic Memory (DM) subsystem; some AutoLISP classes do not use DM for memory allocation.

Return Values

`nil`

Examples

Command: **(mem)**

```
; GC calls: 23; GC run time: 298 ms
Dynamic memory segments statistic:
PgSz  Used  Free  FMCL  Segs  Type
 512    79    48    48     1  lisp stacks
 256  3706   423   142    16  bytecode area
4096   320    10    10    22  CONS memory
  32   769  1213  1089     1  ::new
4096   168    12    10    12  DM Str
4096   222     4     4    15  DMxx memory
 128     4   507   507     1  bstack body
Segment size: 65536, total used: 68, free: 0
nil
```

member

Searches a list for an occurrence of an expression and returns the remainder of the list, starting with the first occurrence of the expression

`(member` ***`expr lst`*****`)`**

Arguments

expr	The expression to be searched for.
lst	The list in which to search for *expr*.

Return Values

A list, or `nil`, if there is no occurrence of *expr* in *lst*.

Examples

Command: **(member 'c '(a b c d e))**
(C D E)

Command: **(member 'q '(a b c d e))**
nil

menucmd

Issues menu commands, or sets and retrieves menu item status

`(menucmd` *`string`*`)`

The **`menucmd`** function can switch between subpages in an AutoCAD menu. This function can also force the display of menus. This allows AutoLISP programs to use image tile menus and to display other menus from which the user can make selections. AutoLISP programs can also enable, disable, and place marks in menu items.

Arguments

string A string that specifies a menu area and the value to assign to that menu area. The *string* argument has the following parameters.

```
"menu_area=value"
```

The allowed values of *menu_area*, shown in the following list, are the same as they are in menu file submenu references. For more information, see “Pull-Down and Shortcut Menus” in the *Customization Guide*.

B1–B4 BUTTONS menus 1 through 4.

A1–A4 AUX menus 1 through 4.

P0–P16 Pull-down (POP) menus 0 through 16.

I Image tile menus.

S SCREEN menu.

T1–T4 TABLET menus 1 through 4.

M DIESEL string expressions.

`Gmenugroup.nametag` A menugroup and name tag.

Return Values

```
nil
```

Examples

The following code displays the image tile menu MOREICONS.

```
(menucmd "I=moreicons")          Loads the MOREICONS image tile menu
(menucmd "I=*")                  Displays the menu
```

The following code checks the status of the third menu item in the pull-down menu POP11. If the menu item is currently enabled, the **menucmd** function disables it.

```
(setq s (menucmd "P11.3=?"))  Gets the status of the menu item
(if (= s "")                  If the status is an empty string,
  (menucmd "P11.3=~")         disable the menu item
)
```

The previous code is not foolproof. In addition to being enabled or disabled, menu items can also receive marks. The code `(menucmd "P11.3=?")` could return `"!."`, indicating that the menu item is currently checked. This code would assume that the menu item is disabled and continue without disabling it. If the code included a call to the **wcmatch** function, it could check the status for an occurrence of the tilde (~) character and then take appropriate action.

The **menucmd** function also allows AutoLISP programs to take advantage of the DIESEL string expression language. Some things can be done much easier with DIESEL than with the equivalent AutoLISP code. The following code returns a string containing the current day and date:

```
(menucmd "M=$(edtime,$(getvar,date),DDDD\",\" D MONTH YYYY)")
 returns   "Sunday, 16 July 1995"
```

See Also

The *Customization Guide* for more information on using AutoLISP to access menu label status, and for information on using DIESEL.

menugroup

Verifies that a menugroup is loaded

```
(menugroup groupname)
```

Arguments

groupname	A string that specifies the menugroup name.

Return Values

If *groupname* matches a loaded menugroup the function returns the *group-name* string; otherwise, it returns `nil`.

min

Returns the smallest of the numbers given

```
(min [number number...])
```

Arguments

number	A number.

Return Values

A number. If any *number* argument is a real, a real is returned, otherwise an integer is returned. If no argument is supplied, **`min`** returns 0.

Examples

Command: **(min 683 -10.0)**
-10.0

Command: **(min 73 2 48 5)**
2

Command: **(min 73.0 2 48 5)**
2.0

Command: **(min 2 4 6.7)**
2.0

minusp

Verifies that a number is negative

```
(minusp num)
```

Arguments

num	A number.

Return Values

`T` if *number* is negative, `nil` otherwise.

Examples

Command: **(minusp -1)**
T

Command: **(minusp -4.293)**
T

Command: **(minusp 830.2)**
nil

mode_tile

Sets the mode of a dialog box tile

```
(mode_tile key mode)
```

Arguments

key	A string that specifies the tile. The key argument is case-sensitive.
mode	An integer that can be one of the following:

0	Enable tile
1	Disable tile
2	Set focus to tile
3	Select edit box contents
4	Flip image highlighting on or off

Return Values

`nil`

namedobjdict

Returns the entity name of the current drawing's named object dictionary, which is the root of all nongraphical objects in the drawing

`(namedobjdict)`

Using the name returned by this function and the dictionary access functions, an application can access the nongraphical objects in the drawing.

nentsel

Prompts the user to select an object (entity) by specifying a point, and provides access to the definition data contained within a complex object

`(nentsel` ***`[msg]`*** **`)`**

The **`nentsel`** function prompts the user to select an object. The current Object Snap mode is ignored unless the user specifically requests it. To provide additional support at the Command prompt, **`nentsel`** honors keywords defined by a previous call to **`initget`**.

Arguments

msg	A string to be displayed as a prompt. If omitted, the Select object prompt is issued.

Return Values

When the selected object is not complex (i.e., not a 3D polyline or block), **`nentsel`** returns the same information as **`entsel`**. However, if the selected object is a 3D polyline, **`nentsel`** returns a list containing the name of the subentity (vertex) and the pick point. This is similar to the list returned by **`entsel`**, except that the name of the selected vertex is returned instead of the polyline header. The **`nentsel`** function always returns the starting vertex of the selected 3D polyline segment. Picking the third segment of the polyline, for example, returns the third vertex. The Seqend subentity is never returned by **`nentsel`** for a 3D polyline.

NOTE A lightweight polyline (lwpolyline entity) is defined in the drawing database as a single entity; it does not contain subentities.

Selecting an attribute within a block reference returns the name of the attribute and the pick point. When the selected object is a component of a block reference other than an attribute, `nentsel` returns a list containing four elements.

The first element of the list returned from picking an object within a block is the selected entity's name. The second element is a list containing the coordinates of the point used to pick the object.

The third element is called the Model to World Transformation Matrix. It is a list consisting of four sublists, each of which contains a set of coordinates. This matrix can be used to transform the entity definition data points from an internal coordinate system called the Model Coordinate System (MCS), to the World Coordinate System (WCS). The insertion point of the block that contains the selected entity defines the origin of the MCS. The orientation of the UCS when the block is created determines the direction of the MCS axes.

NOTE `nentsel` is the only AutoLISP function that uses a matrix of this type; the `nentselp` function returns a matrix similar to those used by other AutoLISP and ObjectARX functions.

The fourth element is a list containing the entity name of the block that contains the selected object. If the selected object is in a nested block (a block within a block), the list additionally contains the entity names of all blocks in which the selected object is nested, starting with the innermost block and continuing outward until the name of the block that was inserted in the drawing is reported.

For information on converting MCS coordinates to WCS, see “Entity Context and Coordinate Transform Data” in the “Using AutoLISP to Manipulate AutoCAD Objects” chapter of the *Visual LISP Developer's Guide*.

Examples

Draw a 3Dpolyline with multiple line segments, then load and run the following function and select different segments of the line. Pick off of the line and then pick the same segments again to see the subentity handle. Try it with a lightweight polyline to see the difference.

```
(defun c:subent ()
  (while
     (setq Ent (entsel "\nPick an entity: "))
     (print (strcat "Entity handle is: "
          (cdr (assoc 5 (entget (car Ent))))))
   )
   (while
      (setq Ent (nentsel "\nPick an entity or subEntity: "))
      (print (strcat "Entity or subEntity handle is:  "
          (cdr (assoc 5 (entget (car Ent))))))
   )
  (prompt "\nDone.")
  (princ)
)
```

See Also

The entsel, initget, and nentselp functions in this reference and “Entity Name Functions” in the *Visual LISP Developer’s Guide*.

nentselp

Provides similar functionality to that of the `nentsel` function without the need for user input

`(nentselp [msg] [pt])`

Arguments

msg	A string to be displayed as a prompt. If omitted, the Select object prompt is issued.
pt	A selection point. This allows object selection without user input.

Return Values

The **`nentselp`** function returns a 4 × 4 transformation matrix, defined as follows:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{bmatrix}$$

The first three columns of the matrix specify scaling and rotation. The fourth column is a translation vector.

The functions that use a matrix of this type treat a point as a column vector of dimension 4. The point is expressed in *homogeneous coordinates,* where the fourth element of the point vector is a *scale factor* that is normally set to 1.0. The final row of the matrix, the vector [M_{30} M_{31} M_{32} M_{33}], has the nominal value of [0 0 0 1]; it is currently ignored by the functions that use this matrix format. In this convention, applying a transformation to a point is a matrix multiplication that appears as follows:

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1.0 \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1.0 \end{bmatrix}$$

This multiplication gives us the individual coordinates of the point as follows:

$$\begin{aligned} X' &= XM_{00} + YM_{01} + ZM_{02} + M_{03}(1.0) \\ Y' &= XM_{10} + YM_{11} + ZM_{12} + M_{13}(1.0) \\ Z' &= XM_{20} + YM_{21} + ZM_{22} + M_{23}(1.0) \end{aligned}$$

As these equations show, the scale factor and the last row of the matrix have no effect and are ignored.

See Also

The nentsel function.

new_dialog

Begins a new dialog box and displays it, and can also specify a default action

```
(new_dialog dlgname dcl_id [action [screen-pt]])
```

Arguments

dlgname	A string that specifies the dialog box.

dcl_id	The DCL file identifier obtained by **`load_dialog`**.
action	A string that contains an AutoLISP expression to use as the default action. If you don't want to define a default action, specify an empty string (`""`). The *action* argument is required if you specify *screen-pt*. The default action is evaluated when the user picks an active tile that doesn't have an action or callback explicitly assigned to it by **`action_tile`** or in DCL.
screen-pt	A 2D point list that specifies the *X,Y* location of the dialog box on the screen. The point specifies the upper-left corner of the dialog box. If you pass the point as `'(-1 -1)`, the dialog box is opened in the default position (the center of the AutoCAD graphics screen).

Return Values

`T`, if successful, otherwise `nil`.

See Also

The "Managing Dialog Boxes" chapter of the *Visual LISP Developer's Guide*.

not

Verifies that an item evaluates to `nil`

`(not` ***`item`*****`)`**

Typically, the **`null`** function is used for lists, and **`not`** is used for other data types along with some types of control functions.

Arguments

item	An AutoLISP expression.

Return Values

`T` if *item* evaluates to `nil`, `nil` otherwise.

Examples

Command: **(setq a 123 b "string" c nil)**
nil

Command: **(not a)**
nil

Command: **(not b)**
nil

Command: **(not c)**
T

Command: **(not '())**
T

See Also

The null function.

nth

Returns the nth element of a list

`(nth n lst)`

Arguments

n	The number of the element to return from the list (zero is the first element).
lst	The list.

Return Values

The *n*th element of *lst*. If *n* is greater than the highest element number of *lst*, **`nth`** returns `nil`.

Examples

Command: **(nth 3 '(a b c d e))**
D

Command: **(nth 0 '(a b c d e))**
A

Command: **(nth 5 '(a b c d e))**
nil

null

Verifies that an item is bound to `nil`

`(null `*`item`*`)`

Arguments

item	An AutoLISP expression.

Return Values

`T` if *item* evaluates to `nil`, `nil` otherwise.

Examples

Command: **(setq a 123 b "string" c nil)**
nil

Command: **(null a)**
nil

Command: **(null b)**
nil

Command: **(null c)**
T

Command: **(null '())**
T

See Also

The not function.

numberp

Verifies that an item is a real number or an integer

```
(numberp item)
```

Arguments

item	An AutoLISP expression.

Return Values

`T` if *item* evaluates to a real or an integer, `nil` otherwise.

Examples

```
Command: (setq a 123 b 'a)
A

Command: (numberp 4)
T

Command: (numberp 3.8348)
T

Command: (numberp "Howdy")
nil

Command: (numberp a)
T

Command: (numberp b)
nil

Command: (numberp (eval b))
T
```

open

Opens a file for access by the AutoLISP I/O functions

```
(open filename mode)
```

Arguments

filename	A string that specifies the name and extension of the file to be opened. If you do not specify the full path name of the file, **open** assumes you are referring to the AutoCAD start-up directory.
mode	Indicates whether the file is open for reading, writing, or appending. Specify a string containing one of the following letters:

r Open for reading.

w Open for writing. If *filename* does not exist, a new file is created and opened. If filename already exists, its existing data is overwritten. Data passed to an open file is not actually written until the file is closed with the **close** function.

a Open for appending. If *filename* does not exist, a new file is created and opened. If *filename* already exists, it is opened and the pointer is positioned at the end of the existing data, so new data you write to the file is appended to the existing data.

The *mode* argument can be uppercase or lowercase. Note that in releases prior to AutoCAD 2000, *mode* had to be specified in lowercase.

Return Values

If successful, **open** returns a file descriptor that can be used by the other I/O functions. If mode `"r"` is specified and *filename* does not exist, **open** returns `nil`.

NOTE On DOS systems, some programs and text editors write text files with an end-of-file marker (CTRL+Z, decimal ASCII code 26) at the end of the text. When reading a text file, DOS returns an end-of-file status if a CTRL+Z marker is encountered, even if that marker is followed by more data. If you intend to use OPEN's `"a"` mode to append data to files produced by another program, be certain the other program does not insert CTRL+Z markers at the end of its text files.

Examples

Open an existing file:

Command: **(setq a (open** "**c:/program files/autocad 2000i/help/filelist.txt**" "**r**"**))**
#<file "c:/program files/autocad 2000i/help/filelist.txt">

The following examples issue **`open`** against files that do not exist:

Command: **(setq f (open** "**c:\\my documents\\new.txt**" "**w**"**))**
#<file "c:\\my documents\\new.tst">

Command: **(setq f (open** "**nosuch.fil**" "**r**"**))**
nil

Command: **(setq f (open** "**logfile**" "**a**"**))**
#<file "logfile">

or

Returns the logical OR of a list of expressions

`(or` ***`[expr...]`*** **`)`**

The **`or`** function evaluates the expressions from left to right, looking for a non-`nil` expression.

Arguments

expr	The expressions to be evaluated.

Return Values

`T`, if a non-`nil` expression is found, or `nil`, if all of the expressions are `nil` or no arguments are supplied.

Note that **`or`** accepts an atom as an argument and returns `T` if one is supplied.

Examples

Command: **(or nil 45 '())**
T

Command: **(or nil '())**
nil

osnap

Returns a 3D point that is the result of applying an Object Snap mode to a specified point

`(osnap` ***`pt mode`*****`)`**

Arguments

pt	A point.
mode	A string that consists of one or more valid Object Snap identifiers such as `mid`, `cen`, and so on, separated by commas.

Return Values

A point, or `nil`, if the pick did not return an object (for example, there is no geometry under the pick aperture, or the geometry is not applicable to the selected object snap mode). The point returned by **`osnap`** depends on the current 3D view, the AutoCAD entity around *pt*, and the setting of the APERTURE system variable.

Examples

Command: **(setq pt1 (getpoint))**
(11.8637 3.28269 0.0)

Command: **(setq pt2 (osnap pt1 "_end,_int"))**
(12.1424 3.42181 0.0)

polar

Returns the UCS 3D point at a specified angle and distance from a point

```
(polar pt ang dist)
```

Arguments

pt	A 2D or 3D point.
ang	An angle expressed in radians relative to the *X* axis, with respect to the current construction plane. Angles increase in the counterclockwise direction.
dist	Distance from the specified *pt*.

Return Values

A 2D or 3D point, depending on the type of point specified by *pt*.

Examples

Supplying a 3D point to **`polar`**:

Command: **(polar '(1 1 3.5) 0.785398 1.414214)**
(2.0 2.0 3.5)

Supplying a 2D point to **`polar`**:

Command: **(polar '(1 1) 0.785398 1.414214)**
(2.0 2.0)

prin1

Prints an expression to the command line or writes an expression to an open file

```
(prin1 [expr [file-desc]])
```

Arguments

expr	A string or AutoLISP expression. Only the specified *expr* is printed; no newline or space is included.
file-desc	A file descriptor for a file opened for writing.

Return Values

The value of the evaluated *expr*. If called with no arguments, `prin1` returns a null symbol.

Used as the last expression in a function, `prin1` without arguments results in a blank line printing when the function completes, allowing the function to exit "quietly."

Examples

Command: **(setq a 123 b '(a))**
(A)

Command: **(prin1 'a)**
AA

The previous command printed A and returned A.

Command: **(prin1 a)**
123123

The previous command printed 123 and returned 123.

Command: **(prin1 b)**
(A)(A)

The previous command printed (A) and returned (A).

Each preceding example is displayed on the screen because no *file-desc* was specified. Assuming that `f` is a valid file-descriptor for a file opened for writing, the following function call writes a string to that file and returns the string:

Command: **(prin1 "Hello" f)**
"Hello"

If *expr* is a string containing control characters, `prin1` expands these characters with a leading \, as shown in the following table:

Control codes

Code	Description
`\\`	\ character
`\"`	" character
`\e`	Escape character

Control codes (*continued*)

Code	Description
`\n`	Newline character
`\r`	Return character
`\t`	TAB character
`\`*nnn*	Character whose octal code is *nnn*

The following example shows how to use control characters:

```
Command: (prin1 (chr 2))
"\002""\002"
```

See Also

The “Displaying Messages” topic in the *Visual LISP Developer’s Guide*.

princ

Prints an expression to the command line, or writes an expression to an open file

(princ *[expr [file-desc]]*)

This function is the same as **`prin1`**, except control characters in *expr* are printed without expansion. In general, **`prin1`** is designed to print expressions in a way that is compatible with **`load`**, while **`princ`** prints them in a way that is readable by functions such as **`read-line`**.

Arguments

expr	A string or AutoLISP expression. Only the specified *expr* is printed; no newline or space is included.
file-desc	A file descriptor for a file opened for writing.

Return Values

The value of the evaluated *expr*. If called with no arguments, **`princ`** returns a null symbol.

See Also

The “Displaying Messages” topic in the *Visual LISP Developer’s Guide*.

print

Prints an expression to the command line, or writes an expression to an open file

```
(print [expr [file-desc]])
```

This function is the same as **`prin1`**, except it prints a newline character before *expr*, and prints a space following *expr*.

Arguments

expr	A string or AutoLISP expression. Only the specified *expr* is printed; no newline or space is included.
file-desc	A file descriptor for a file opened for writing.

Return Values

The value of the evaluated *expr*. If called with no arguments, **`print`** returns a null symbol.

See Also

The "Displaying Messages" topic in the *Visual LISP Developer's Guide*.

progn

Evaluates each expression sequentially and returns the value of the last expression

```
(progn [expr]...)
```

You can use **`progn`** to evaluate several expressions where only one expression is expected.

Arguments

expr	One or more AutoLISP expressions.

Return Values

The result of the last evaluated expression.

Examples

The **`if`** function normally evaluates one *then* expression if the test expression evaluates to anything but `nil`. The following example uses **`progn`** to evaluate two expressions following **`if`**:

```
(if (= a b)
  (progn
    (princ "\nA = B ")
    (setq a (+ a 10) b (- b 10))
  )
)
```

See Also

The if function.

prompt

Displays a string on your screen's prompt area

`(prompt` ***`msg`*****`)`**

On dual-screen AutoCAD configurations, **`prompt`** displays *msg* on both screens and is, therefore, preferable to **`princ`**.

Arguments

msg	A string.

Return Values

```
nil
```

Examples

Command: **(prompt** "**New value:** "**)**
New value: nil

See Also

The "Displaying Messages" topic in the *Visual LISP Developer's Guide*.

quit

Forces the current application to quit

```
(quit)
```

If `quit` is called, it returns the error message quit/exit abort and returns to the AutoCAD Command prompt.

See Also

The exit function.

quote

Returns an expression without evaluating it

```
(quote expr)
```

Arguments

expr	An AutoLISP expression.

Return Values

The *expr* argument.

Examples

Command: **(quote a)**
A

The previous expression can also be written as `'a`. For example:

Command: **!'a**
A

Command: **(quote (a b))**
(A B)

See Also

The function function.

read

Returns the first list or atom obtained from a string

```
(read [string])
```

The **`read`** function parses the string representation of any LISP data and returns the first expression in the string, converting it to a corresponding data type.

Arguments

string	A string. The *string* argument should not contain blanks, except within a list or string.

Return Values

A list or atom. The **`read`** function returns its argument converted into the corresponding data type. If no argument is specified, **`read`** returns `nil`.

If the string contains multiple LISP expressions separated by LISP symbol delimiters such as blanks, new-line, tabs, or parentheses, only the first expression is returned.

Examples

```
Command: (read "hello")
HELLO

Command: (read "hello there")
HELLO

Command: (read "\"Hi Y'all\"")
"Hi Y'all"

Command: (read "(a b c)")
(A B C)

Command: (read "(a b c) (d)")
(A B C)

Command: (read "1.2300")
1.23
```

Command: **(read "87")**
87

Command: **(read "87 3.2")**
87

read-char

Returns the decimal ASCII code representing the character read from the keyboard input buffer or from an open file

```
(read-char [file-desc])
```

Arguments

file-desc	A file descriptor (obtained from `open`) referring to an open file. If no *file-desc* is specified, `read-char` obtains input from the keyboard input buffer.

Return Values

An integer representing the ASCII code for a character. The `read-char` function returns a single newline character (ASCII code 10) whenever it detects an end-of-line character or character sequence.

Examples

The following example omits *file-desc*, so `read-char` looks for data in the keyboard buffer:

Command: **(read-char)**

The keyboard buffer is empty, so `read-char` waits for user input.

```
ABC
65
```

The user entered `ABC`; `read-char` returned the ASCII code representing the first character entered (A). The next three calls to `read-char` return the data remaining in the keyboard input buffer. This data translates to 66 (the ASCII code for the letter B), 67 (C), and 10 (newline), respectively:

```
Command: (read-char)
66
```

```
Command: (read-char)
67
```

```
Command: (read-char)
10
```

With the keyboard input buffer now empty, **`read-cha`**r waits for user input the next time it is called:

```
Command: (read-char)
```

read-line

Reads a string from the keyboard or from an open file, until an end-of-line marker is encountered

`(read-line [file-desc])`

Arguments

file-desc	A file descriptor (obtained from **`open`**) referring to an open file. If no *file-desc* is specified, **`read-line`** obtains input from the keyboard input buffer.

Return Values

The string read by **`read-line`**, without the end-of-line marker. If **`read-line`** encounters the end of the file, it returns `nil`.

Examples

Open a file for reading:

```
Command: (setq f (open "c:\\my documents\\new.tst" "r"))
#<file "c:\\my documents\\new.tst">
```

Use **`read-line`** to read a line from the file:

```
Command: (read-line f)
"To boldly go where nomad has gone before."
```

Obtain a line of input from the user:

Command: **(read-line)**
To boldly go
"To boldly go"

redraw

Redraws the current viewport or a specified object (entity) in the current viewport

```
(redraw [ename [mode]])
```

If **`redraw`** is called with no arguments, the function redraws the current viewport. If called with an entity name argument, **`redraw`** redraws the specified entity.

The **`redraw`** function has no effect on highlighted or hidden entities, however a REGEN command forces the entities to redisplay in their normal manner.

Arguments

ename	The name of the entity name to be redrawn.
mode	An integer value that controls the visibility and highlighting of the entity. The *mode* can be one of the following values:

1 Show entity

2 Hide entity (blank it out)

3 Highlight entity

4 Unhighlight entity

The use of entity highlighting (mode 3) must be balanced with entity unhighlighting (mode 4).

If *ename* is the header of a complex entity (a polyline or a block reference with attributes), **`redraw`** processes the main entity and all its subentities if the *mode* argument is positive. If the *mode* argument is negative, **`redraw`** operates on only the header entity.

Return Values

The **`redraw`** function always returns `nil`.

regapp

Registers an application name with the current AutoCAD drawing in preparation for using extended object data

`(regapp` *`application`*`)`

Arguments

application	A string naming the application. The name must be a valid symbol table name. See the description of snvalid for the rules AutoLISP uses to determine if a symbol name is valid.

Return Values

If an application of the same name has already been registered, this function returns `nil`; otherwise it returns the name of the application.

If registered successfully, the application name is entered into the APPID symbol table. This table maintains a list of the applications that are using extended data in the drawing.

Examples

```
(regapp "ADESK_4153322344")
(regapp "DESIGNER-v2.1-124753")
```

NOTE It is recommended that you pick a unique application name. One way of ensuring this is to adopt a naming scheme that uses the company or product name and a unique number (like your telephone number or the current date/time). The product version number can be included in the application name or stored by the application in a separate integer or real-number field; for example, `(1040 2.1)`.

rem

Divides the first number by the second, and returns the remainder

(rem *[number number...]*)

Arguments

number	Any number. If you provide more than two numbers, **rem** divides the result of dividing the first number by the second with the third, and so on. If you provide more than two numbers, **rem** evaluates the arguments from left to right. For example, if you supply three numbers, **rem** divides the first number by the second, then takes the result and divides it by the third number, returning the remainder of that operation.

Return Values

A number. If any *number* argument is a real, **rem** returns a real, otherwise **rem** returns an integer. If no arguments are supplied, **rem** returns 0. If a single *number* argument is supplied, **rem** returns *number*.

Examples

Command: **(rem 42 12)**
6

Command: **(rem 12.0 16)**
12.0

Command: **(rem 26 7 2)**
1

repeat

Evaluates each expression a specified number of times, and returns the value of the last expression

```
(repeat int [expr...])
```

Arguments

int	An integer. Must be a positive number.
expr	One or more atoms or expressions.

Return Values

The value of the last expression or atom evaluated. If *expr* is not supplied, **repeat** returns nil.

Examples

```
Command: (setq a 10 b 100)
100
```

```
Command: (repeat 4  (setq a (+ a 10)) (setq b (+ b 100)))
500
```

After evaluation, a is 50, b is 500, and **repeat** returns 500.

If strings are supplied as arguments, **repeat** returns the last string:

```
Command: (repeat 100 "Me" "You")
"You"
```

reverse

Returns a copy of a list with its elements reversed

```
(reverse lst)
```

Arguments

lst	A list.

Return Values

A list.

Examples

Command: **(reverse '((a) b c))**
(C B (A))

rtos

Converts a number into a string

```
(rtos number [mode [precision]])
```

The **`rtos`** function returns a string that is the representation of *number* according to the settings of *mode*, *precision*, and the system variables UNIT-MODE, DIMZIN, LUNITS, and LUPREC.

Arguments

number	A number.
mode	An integer specifying the linear units mode. The *mode* corresponds to the values allowed for the AutoCAD system variable lunits and can be one of the following numbers: **1** Scientific **2** Decimal **3** Engineering (feet and decimal inches) **4** Architectural (feet and fractional inches) **5** Fractional
precision	An integer specifying the precision.

The *mode* and *precision* arguments correspond to the system variables LUNITS and LUPREC. If you omit the arguments, **`rtos`** uses the current settings of LUNITS and LUPREC.

Return Values

A string. The UNITMODE system variable affects the returned string when engineering, architectural, or fractional units are selected (*mode* values 3, 4, or 5).

Examples

Set variable `x`:

```
Command: (setq x 17.5)
17.5
```

Convert the value of `x` to a string in scientific format, with a precision of 4:

```
Command: (setq fmtval (rtos x 1 4))
"1.7500E+01"
```

Convert the value of `x` to a string in decimal format, with 2 decimal places:

```
Command: (setq fmtval (rtos x 2 2))
"17.50"
```

Convert the value of `x` to a string in engineering format, with a precision of 2:

```
Command: (setq fmtval (rtos x 3 2))
"1'-5.50\""
```

Convert the value of `x` to a string in architectural format:

```
Command: (setq fmtval (rtos x 4 2))
"1'-5 1/2\""
```

Convert the value of `x` to a string in fractional format:

```
Command: (setq fmtval (rtos x 5 2))
"17 1/2"
```

Setting UNITMODE to 1 causes units to be displayed as entered. This affects the values returned by **`rtos`** for engineering, architectural, and fractional formats, as shown in the following examples:

```
Command: (setvar "unitmode" 1)
1

Command: (setq fmtval (rtos x 3 2))
"1'5.50\""

Command: (setq fmtval (rtos x 4 2))
"1'5-1/2\""

Command: (setq fmtval (rtos x 5 2))
"17-1/2"
```

See Also

The “String Conversions” topic in the *Visual LISP Developer’s Guide*.

set

Sets the value of a quoted symbol name to an expression

(set *sym expr*)

The **set** function is similar to **setq** except that **set** evaluates both of its arguments whereas **setq** only evaluates its second argument.

Arguments

sym	A symbol.
expr	An AutoLISP expression.

Return Values

The value of the expression.

Examples

Each of the following commands sets symbol a to 5.0:

```
(set 'a 5.0)
(set (read "a") 5.0)
(setq a 5.0)
```

Both **set** and **setq** expect a symbol as their first argument, but **set** accepts an expression that returns a symbol, whereas **setq** does not, as the following shows:

Command: **(set (read "a") 5.0)**
5.0

Command: **(setq (read "a") 5.0)**
; *** ERROR: syntax error

See Also

The setq function.

set_tile

Sets the value of a dialog box tile

`(set_tile key value)`

Arguments

key	A string that specifies the tile.
value	A string that names the new value to assign (initially set by the `value` attribute).

Return Values

The *value* the tile was set to.

setcfg

Writes application data to the AppData section of the *acad.cfg* file

`(setcfg cfgname cfgval)`

Arguments

cfgname	A string that specifies the section and parameter to set with the value of *cfgval*. The *cfgname* argument must be a string of the following form: AppData/*application_name*/*section_name*/.../*param_name* The string can be up to 496 characters long.
cfgval	A string. The string can be up to 512 characters in length. Larger strings are accepted by **`setcfg`**, but cannot be returned by **`getcfg`**.

Return Values

If successful, **`setcfg`** returns *cfgval*. If *cfgname* is not valid, **`setcfg`** returns `nil`.

Examples

The following code sets the WallThk parameter in the AppData/ArchStuff section to 8, and returns the string “8”:

```
Command: (setcfg "AppData/ArchStuff/WallThk" "8")
"8"
```

See Also

The getcfg function.

setenv

Sets a system environment variable to a specified value

(setenv *varname value*)

Arguments

varname	A string specifying the name of the environment variable to be set. Environment variable names must be spelled and cased exactly as they are stored in the system registry.
value	A string specifying the value to set *varname* to.

Return Values

value

Examples

The following command sets the value of the `MaxArray` environment variable to 10000:

```
Command: (setenv "MaxArray" "10000")
"10000"
```

Note that changes to settings might not take effect until the next time AutoCAD is started.

See Also

The getenv function.

setfunhelp

Registers a user-defined command with the Help facility so the appropriate help file and topic are called when the user requests help on that command

```
(setfunhelp c:fname [helpfile [topic [command]]])
```

Arguments

c:fname	A string specifying the user-defined command (the **`C:`*`XXX`*** function). You must include the **`c:`** prefix.
helpfile	A string naming the help file. The file extension is not required with the *helpfile* argument. If a file extension is provided, AutoCAD looks only for a file with the exact name specified. If no file extension is provided, AutoCAD looks for *helpfile* with an extension of *.chm*. If no file of that name is found, AutoCAD looks for file with an extension of *.hlp*.
topic	A string identifying a Help topic ID. If you are calling a topic within a CHM file, provide the file name without the extension; AutoCAD adds an *.htm* extension.
command	A string that specifies the initial state of the Help window. The *command* argument is a string used by the uCommand (in HTML Help) or the fuCommand (in WinHelp) argument of the HtmlHelp() and WinHelp() functions as defined in the Microsoft Windows SDK. For HTML Help files, the *command* parameter can be HH_ALINK_LOOKUP or HH_DISPLAY_TOPIC. For Windows Help files, the *command* parameter can be HELP_CONTENTS, HELP_HELPONHELP, or HELP_PARTIALKEY.

Return Values

The string passed as *c:fname*, if successful, otherwise, `nil`.

This function verifies only that the *c:fname* argument has the `c:` prefix. It does *not* verify that the **`c:fname`** function exists, nor does it verify the correctness of the other arguments supplied.

Examples

The following example illustrates the use of **`setfunhelp`** by defining a simple function and issuing **`setfunhelp`** to associate the function with the *circle* topic in the AutoCAD help file (*acad.chm*):

```
(defun c:foo ()
  (getstring "Press F1 for help on the foo command:")
)
(setfunhelp "c:foo" "acad.chm" "circle")
```

After loading this code, issuing the **`foo`** command and then pressing F1 displays the circle topic.

This example works, but serves no real purpose. In the real world, you would create your own help file and associate that help file and topic with your function.

Define a function named **`test`**:

Command: **(defun c:test()(getstring** "**\nTEST:** " **)(princ))**
C:TEST

Associate the function with a call to help with the string “line”:

Command: **(setfunhelp** "**c:test**" "**acad.chm**" "**line**"**)**
"c:test"

Run the **`test`** command and at the prompt, press F1; you should see the Help topic for the AutoCAD LINE command.

NOTE When you use the **`defun`** function to define a **`C:`*`XXX`*** function, it removes that function’s name from those registered by **`setfunhelp`** (if one exists). Therefore, **`setfunhelp`** should only be called after the **`defun`** call, which defines the user-defined command.

See Also

The defun and help functions.

setq

Sets the value of a symbol or symbols to associated expressions

(setq sym expr [sym expr]...)

This is the basic assignment function in AutoLISP. The **setq** function can assign multiple symbols in one call to the function.

Arguments

sym	A symbol. This argument is not evaluated.
expr	An expression.

Return Values

The result of the last *expr* evaluated.

Examples

The following function call set variable a to 5.0:

```
Command: (setq a 5.0)
5.0
```

Whenever a is evaluated, it returns the real number 5.0.

The following command sets two variables, b and c:

```
Command: (setq b 123 c 4.7)
4.7
```

setq returns the value of the last variable set.

In the following example, s is set to a string:

```
Command: (setq s "it")
"it"
```

The following example assigns a list to x:

```
Command: (setq x '(a b))
(A B)
```

See Also

The “AutoLISP Variables” topic in the *Visual LISP Developer’s Guide*.

setvar

Sets an AutoCAD system variable to a specified value

```
(setvar varname value)
```

Arguments

varname	A string or symbol naming a variable.
value	An atom or expression whose evaluated result is to be assigned to *varname*. For system variables with integer values, the supplied *value* must be between –32,768 and +32,767.

Return Values

If successful, **`setvar`** returns *value*.

Examples

Set the AutoCAD fillet radius to 0.5 units:

```
Command: (setvar "FILLETRAD" 0.50)
0.5
```

Notes on Using setvar

Some AutoCAD commands obtain the values of system variables before issuing any prompts. If you use **`setvar`** to set a new value while a command is in progress, the new value might not take effect until the next AutoCAD command.

When using the **`setvar`** function to change the AutoCAD system variable ANGBASE, the value argument is interpreted as radians. This differs from the AutoCAD SETVAR command, which interprets this argument as degrees. When using the **`setvar`** function to change the AutoCAD system variable SNAPANG, the value argument is interpreted as radians relative to the AutoCAD default direction for angle 0, which is *east* or *3 o'clock*. This also differs from the SETVAR command, which interprets this argument as degrees relative to the ANGBASE setting.

NOTE The UNDO command does not undo changes made to the CVPORT system variable by the **`setvar`** function.

You can find a list of the current AutoCAD system variables in the *Command Reference*.

See Also

The getvar function.

setview

Establishes a view for a specified viewport

```
(setview view_descriptor [vport_id])
```

Arguments

view_descriptor	An entity definition list similar to that returned by **`tblsearch`** when applied to the VIEW symbol table.
vport_id	An integer identifying the viewport to receive the new view. If *vport_id* is 0, the current viewport receives the new view. You can obtain the *vport_id* number from the CVPORT system variable.

Return Values

If successful, the **`setview`** function returns the *view_descriptor*.

sin

Returns the sine of an angle as a real number expressed in radians

```
(sin ang)
```

Arguments

ang	An angle, in radians.

Return Values

A real number representing the sine of *ang*, in radians.

Examples

Command: **(sin 1.0)**
0.841471

Command: **(sin 0.0)**
0.0

slide_image

Displays an AutoCAD slide in the currently active dialog box image tile

```
(slide_image x1 y1 width height sldname)
```

Arguments

x1	*X*-offset from the upper-left corner of the tile, in pixels. Must be a positive value.
y1	*Y*-offset from the upper-left corner of the tile, in pixels. Must be a positive value.
width	Width of the image, in pixels.
height	Height of the image, in pixels.
sldname	Identifies the slide. This argument can be a slide file (*.sld*) or a slide in a slide library file (*.slb*). Specify *sldname* the same way you would specify it for the VSLIDE command or for a menu file (see the "Creating Images" topic in the *Visual LISP Developer's Guide*). Use one of the following formats for *sldname*: *sldname or libname*(*sldname*)

The first (upper-left) corner of the slide—its insertion point—is located at (*x1*,*y1*), and the second (lower-right) corner is located at the relative distance (*wid*,*hgt*) from the first (*wid* and *hgt* must be positive values). The origin (0,0) is the upper-left corner of the image. You obtain the coordinates of the lower-right corner by calling the dimension functions (**`dimx_tile`** and **`dimy_tile`**).

Return Values

A string containing *sldname*.

Examples

```
(slide_image
  0
  0
  (dimx_tile "slide_tile")
  (dimy_tile "slide_tile")
  "myslide"
)
(end_image)
```

snvalid

Checks the symbol table name for valid characters

`(snvalid` ***`sym_name [flag]`*****`)`**

The **`snvalid`** function inspects the system variable EXTNAMES to determine the rules to enforce for the active drawing. If EXTNAMES is 0, **`snvalid`** validates using the symbol name rules in effect prior to AutoCAD 2000. If EXTNAMES is 1 (the default value), **`snvalid`** validates using the rules for extended symbol names introduced with AutoCAD 2000. The following are not allowed in any symbol names, regardless of the setting of EXTNAMES:

- Control and graphic characters
- Null strings
- Vertical bars as the first or last character of the name

AutoLISP does not enforce restrictions on the length of symbol table names if extnames is 1.

Arguments

sym_name	A string that specifies a symbol table name.
flag	An integer that specifies whether the vertical bar character is allowed within *sym_name*. The *flag* argument can be one of the following: **0** Do not allow vertical bar characters anywhere in *sym_name*. This is the default. **1** Allow vertical bar characters in *sym_name*, as long as they are not the first or last characters in the name.

Return Values

`T`, if *sym_name* is a valid symbol table name, otherwise `nil`.

If extnames is 1, all characters are allowed except control and graphic characters and the following:

Characters disallowed in symbol table names	
< >	less-than and greater-than symbol
/ \	forward slash and backslash
"	quotation mark
:	colon
?	question mark
*	asterisk
|	vertical bar
,	comma
=	equal sign
`	backquote
;	Semi-colon (ASCII 59)

A symbol table name may contain spaces.

If extnames is 0, symbol table names can consist of upper- and lowercase alphabetic letters (e.g., A–Z), numeric digits (e.g., 0–9), and the dollar sign ($), underscore (_), and hyphen (–) characters.

Examples

The following examples assume EXTNAMES is set to 1:

Command: **(snvalid "hocus-pocus")**
T

Command: **(snvalid "hocus pocus")**
T

Command: **(snvalid "hocus%pocus")**
T

The following examples assume EXTNAMES is set to 0:

Command: **(snvalid "hocus-pocus")**
T

Command: **(snvalid "hocus pocus")**
nil

Command: **(snvalid "hocus%pocus")**
nil

The following example includes a vertical bar in the symbol table name:

Command: **(snvalid "hocus|pocus")**
nil

By default, the vertical bar character is considered invalid in all symbol table names.

In the following example, the *flag* argument is set to 1, so **`snvalid`** considers the vertical bar character to be valid in *sym_name*, as long as it is not the first or last character in the name:

Command: **(snvalid "hocus|pocus" 1)**
T

sqrt

Returns the square root of a number as a real number

`(sqrt` ***`num`*****`)`**

Arguments

num	A number (integer or real).

Return Values

A real number.

Examples

Command: **(sqrt 4)**
2.0

Command: **(sqrt 2.0)**
1.41421

ssadd

Adds an object (entity) to a selection set, or creates a new selection set

```
(ssadd [ename [ss]])
```

Arguments

ename	An entity name.
ss	A selection set.

If called with no arguments, **`ssadd`** constructs a new selection set with no members. If called with the single entity name argument *`ename`*, **`ssadd`** constructs a new selection set containing that single entity. If called with an entity name and the selection set *ss*, **`ssadd`** adds the named entity to the selection set.

Return Values

The new or modified selection set.

Examples

When adding an entity to a set, the new entity is added to the existing set, and the set passed as *ss* is returned as the result. Thus, if the set is assigned to other variables, they also reflect the addition. If the named entity is already in the set, the **`ssadd`** operation is ignored and no error is reported.

Set `e1` to the name of the first entity in drawing:

```
Command: (setq e1 (entnext))
<Entity name: 1d62d60>
```

Set `ss` to a null selection set:

```
Command: (setq ss (ssadd))
<Selection set: 2>
```

The following command adds the `e1` entity to the selection set referenced by `ss`:

```
Command: (ssadd e1 ss)
<Selection set: 2>
```

Get the entity following `e1`:

```
Command: (setq e2 (entnext e1))
<Entity name: 1d62d68>
```

Add `e2` to the `ss` entity:

```
Command: (ssadd e2 ss)
<Selection set: 2>
```

ssdel

Deletes an object (entity) from a selection set

`(ssdel` ***`ename ss`*****`)`**

Arguments

ename	An entity name.
ss	A selection set.

Return Values

The name of the selection set, or `nil`, if the specified entity is not in the set.

Note that the entity is actually deleted from the existing selection set, as opposed to a new set being returned with the element deleted.

Examples

In the following examples, entity name `e1` is a member of selection set `ss`, while entity name `e3` is not a member of `ss`:

```
Command: (ssdel e1 ss)
<Selection set: 2>
```

Selection set `ss` is returned with entity `e1` removed.

```
Command: (ssdel e3 ss)
nil
```

The function returns `nil` because `e3` is not a member of selection set `ss`.

ssget

Creates a selection set from the selected object

(ssget [sel-method] [pt1 [pt2]] [pt-list] [filter-list])

Selection sets can contain objects from both paper and model space, but when the selection set is used in an operation, **ssget** filters out objects from the space not currently in effect. Selection sets returned by **ssget** contain main entities only (no attributes or polyline vertices).

Arguments

sel-method	A string that specifies the object selection method. Valid selection methods are:
	C Crossing selection.
	CP Cpolygon selection (all objects crossing and inside of the specified polygon).
	F Fence selection.
	I Implied selection (objects selected while PICKFIRST is in effect).
	L Last visible object added to the database.
	P Last selection set created.
	W Window selection.
	WP WPolygon (all objects within the specified polygon).
	X Entire database. If you specify the x selection method and do not provide a *filter-list*, **ssget** selects all entities in the database, including entities on layers that are off, frozen, and out of the visible screen.
	:E Everything within the cursor's object selection pickbox.
	:N Call **ssnamex** for additional information on container blocks and transformation matrices for any entities selected during the **ssget** operation. This additional information is available only for entities selected via graphical selection methods such as Window, Crossing, and point picks.

Unlike the other object selection methods, **:N** may return multiple entities with the same entity name in the selection set. For example, if the user selects a subentity of a complex entity such as a BlockReference, PolygonMesh, or old style polyline, **`ssget`** looks at the subentity that is selected when determining if it has already been selected. However, **`ssget`** actually adds the main entity (BlockReference, PolygonMesh, etc.) to the selection set. The result could be multiple entries with the same entity name in the selection set (each will have different subentity information for **`ssnamex`** to report).

:S Allow single selection only.

pt1	A point relative to the selection.
pt2	A point relative to the selection.
pt-list	A list of points.
filter-list	An association list that specifies object properties. Objects that match the *filter-list* are added to the selection set.

If you omit all arguments, **`ssget`** prompts the user with the Select objects prompt, allowing interactive construction of a selection set.

If you supply a point but do not specify an object selection method, AutoCAD assumes the user is selecting an object by picking a single point.

Return Values

The name of the created selection set, if successful, or `nil`, if no objects were selected.

Notes on the Object Selection Methods

- When using the `:N` selection method, if the user selects a subentity of a complex entity such as a BlockReference, PolygonMesh, or old style polyline, **`ssget`** looks at the subentity that is selected when determining if it has already been selected. However, **`ssget`** actually adds the main entity (BlockReference, PolygonMesh, etc.) to the selection set. It is therefore possible to have multiple entries with the same entity name in the selection set (each will have different subentity information for **`ssnamex`** to report). Because the `:N` method does not guarantee that each entry will be unique, code that relies on uniqueness should not use selection sets created using this option.
- When using the `L` selection method in an MDI environment, you cannot always count on the last object drawn to remain visible. For example, if

your application draws a line, and the user subsequently minimizes or cascades the AutoCAD drawing window, the line may no longer be visible. If this occurs, **`ssget`** with the `"L"` option will return `nil`.

Examples

Prompt the user to select the objects to be placed in a selection set:

Command: **(ssget)**
<Selection set: 2>

Create a selection set of the object passing through (2,2):

Command: **(ssget '(2 2))**
nil

Create a selection set of the most recently selected objects:

Command: **(ssget "_P")**
<Selection set: 4>

Create a selection set of the objects crossing the box from (0,0) to (1,1):

Command: **(ssget "_C" '(0 0) '(1 1))**
<Selection set: b>

Create a selection set of the objects inside the window from (0,0):

Command: **(ssget "_W" '(0 0) '(5 5))**
<Selection set: d>

By specifying filters, you can obtain a selection set that includes all objects of a given type, on a given layer, or of a given color. The following example returns a selection set that consists only of blue lines that are part of the implied selection set (those objects selected while PICKFIRST is in effect):

Command: **(ssget "_I" '((0 . "LINE") (62 . 5)))**
<Selection set: 4>

The following examples of **`ssget`** require that a list of points be passed to the function. The `pt_list` variable cannot contain points that define zero-length segments.

Create a list of points:

Command: **(setq pt_list '((1 1)(3 1)(5 2)(2 4)))**
((1 1) (3 1) (5 2) (2 4))

Create a selection set of all objects crossing and inside the polygon defined by *pt_list*:

Command: **(ssget "_CP" pt_list)**
<Selection set: 13>

Create a selection set of all blue lines inside the polygon defined by *pt_list*:

Command: **(ssget "_WP" pt_list '((0 . "LINE") (62 . 5)))**
<Selection set: 8>

The selected objects are highlighted only when **`ssget`** is used with no arguments. Selection sets consume AutoCAD temporary file slots, so AutoLISP is not permitted to have more than 128 open at one time. If this limit is reached, AutoCAD refuses to create any more selection sets and returns `nil` to all **`ssget`** calls. To close an unnecessary selection set variable, set it to `nil`.

A selection set variable can be passed to AutoCAD in response to any Select objects prompt at which selection by Last is valid. It selects all the objects in the selection set variable.

The current setting of Object Snap mode is ignored by **`ssget`** unless you specifically request it while you are in the function.

See Also

The "Selection Set Handling" and "Selection Set Filter Lists" topics in the *Visual LISP Developer's Guide*.

ssgetfirst

Determines which objects are selected and gripped

`(ssgetfirst)`

Returns a list of two selection sets similar to those passed to **`sssetfirst`**. The first element in the list is a selection set of entities that are gripped but not selected. The second element is a selection set of entities that are both gripped and selected. Either (or both) elements of the list can be `nil`.

NOTE Only entities from the current drawing's model space and paper space, not nongraphical objects or entities in other block definitions, can be analyzed by this function.

See Also

The ssget and sssetfirst functions.

sslength

Returns an integer containing the number of objects (entities) in a selection set

```
(sslength ss)
```

Arguments

ss	A selection set.

Return Values

An integer.

Examples

Add the last object to a new selection set:

Command: **(setq sset (ssget "L"))**
<Selection set: 8>

Use **`sslength`** to determine the number of objects in the new selection set:

Command: **(sslength sset)**
1

ssmemb

Tests whether an object (entity) is a member of a selection set

```
(ssmemb ename ss)
```

Arguments

ename	An entity name.
ss	A selection set.

Return Values

If *ename* is a member of *ss*, **`ssmemb`** returns the entity name. If *ename* is not a member, **`ssmemb`** returns `nil`.

Examples

In the following examples, entity name `e2` is a member of selection set `ss`, while entity name `e1` is not a member of `ss`:

```
Command: (ssmemb e2 ss)
<Entity name: 1d62d68>
```

```
Command: (ssmemb e1 ss)
nil
```

ssname

Returns the object (entity) name of the indexed element of a selection set

`(ssname` *`ss index`*`)`

Entity names in selection sets obtained with **`ssget`** are always names of main entities. Subentities (attributes and polyline vertices) are not returned. (The **`entnext`** function allows access to them.)

Arguments

ss	A selection set.
index	An integer (or real) indicating an element in a selection set. The first element in the set has an index of zero. To access entities beyond the 32767th one in a selection set, you must supply the *index* argument as a real.

Return Values

An entity name, if successful. If *index* is negative or greater than the highest numbered entity in the selection set, **`ssname`** returns `nil`.

Examples

Get the name of the first entity in a selection set:

```
Command: (setq ent1 (ssname ss 0))
<Entity name: 1d62d68>
```

Get the name of the fourth entity in a selection set:

```
Command: (setq ent4 (ssname ss 3))
<Entity name: 1d62d90>
```

To access entities beyond the 32767th one in a selection set, you must supply the *index* argument as a real, as in the following example:

```
(setq entx (ssname sset 50843.0))
```

See Also

The entnext function.

ssnamex

Retrieves information about how a selection set was created

```
(ssnamex ss [index])
```

Only selection sets with entities from the current drawing's model space and paper space—*not* nongraphical objects or entities in other block definitions—can be retrieved by this function.

Arguments

ss	A selection set.
index	An integer (or real) indicating an element in a selection set. The first element in the set has an index of zero.

Return Values

If successful, **ssnamex** returns the name of the entity at *index*, along with data describing how the entity was selected. If the *index* argument is not supplied, this function returns a list containing the entity names of all of the elements in the selection set, along with data that describes how each entity was selected. If *index* is negative or greater than the highest numbered entity in the selection set, **ssnamex** returns `nil`.

The data returned by **ssnamex** takes the form of a list of lists that contains information that either describes an entity and its selection method or a polygon that was used to select one or more entities. Each sublist that describes the selection of a particular entity comprises three parts: the selection method ID (an integer >= 0), the entity name of the selected entity, and selection method specific data that describes how the entity was selected.

```
((sel_id1 ename1 (data))(sel_id2 ename2 (data)) ... )
```

The following table lists the selection method IDs:

Selection method IDs

ID	Description
0	nonspecific (i.e., Last All etc.)
1	Pick
2	Window or WPolygon
3	Crossing or CPolygon
4	Fence

Each sublist that describes a polygon and is used during entity selection takes the form of a polygon ID (an integer < 0), followed by point descriptions.

```
(polygon_id point_description_1 point_description_n... )
```

Polygon ID numbering starts at –1 and each additional polygon ID is incremented by –1. Depending on the viewing location, a point is represented as one of the following: an infinite line, a ray, or a line segment. A point descriptor comprises three parts: a point descriptor ID (the type of item being described), the start point of the item, and an optional unit vector that describes either the direction in which the infinite line travels or a vector that describes the offset to the other side of the line segment.

```
(point_descriptor_id base_point [unit_or_offset_vector])
```

The following table lists the valid point descriptor IDs:

Point descriptor IDs

ID	Description
0	Infinite line
1	Ray
2	Line segment

The *unit_or_offset_vector* is returned when the view point is something other than 0,0,1.

Examples

The *data* associated with Pick (type 1) entity selections is a single point description. For example, the following record is returned for the selection of an entity picked at 1,1 in plan view of the WCS:

```
Command: (ssnamex ss3 0)
((1 <Entity name: 1d62da0> 0 (0 (1.0 1.0 0.0))))
```

The *data* associated with an entity selected with the Window, WPolygon, Crossing, or CPolygon method is the integer ID of the polygon that selected the entity. It is up to the application to associate the polygon identifiers and make the connection between the polygon and the entities it selected. For example, the following returns an entity selected by Crossing (note that the polygon ID is –1):

```
Command: (ssnamex ss4 0)
((3 <Entity name: 1d62d60> 0 -1) (-1 (0 (-1.80879 8.85536 0.0)) (0 (13.4004
8.85536 0.0)) (0 (13.4004 1.80024 0.0)) (0 (-1.80879 1.80024 0.0))))
```

The data associated with Fence selections is a list of points and descriptions for the points where the fence and entity visually intersect. For example, the following command returns information for a nearly vertical line intersected three times by a Z-shaped fence:

```
Command: (ssnamex ss5 0)
((4 <Entity name: 1d62d88> 0 (0 (5.28135 6.25219 0.0) ) (0 (5.61868 2.81961
0.0) ) (0 (5.52688 3.75381 0.0) ) ) )
```

sssetfirst

Sets which objects are selected and gripped

(sssetfirst *gripset [pickset]*)

The selection set of objects specified by the *gripset* argument are gripped, and the selection set of objects specified by *pickset* are both gripped and selected. If any objects are common to both selection sets, **sssetfirst** grips and selects the selection set specified by *pickset* only (it does not grip the *gripset* set).

You are responsible for creating a valid selection set. For example, you may need to verify that a background paper space viewport (DXF group code 69) is not included in the selection set. You may also need to ensure that selected objects belong to the current layout, as in the following code:

```
(setq ss (ssget (list (cons 410 (getvar "ctab")))))
```

Arguments

gripset	A selection set to be gripped. If *gripset* is `nil` and *pickset* is specified, **`sssetfirst`** grips and selects *pickset*. If *gripset* is `nil` and no *pickset* is specified, **`sssetfirst`** turns off the grip handles and selections it previously turned on.
pickset	A selection set to be selected.

Return Values

The selection set or sets specified.

Examples

First, draw a square and build three selection sets. Begin by drawing side 1 and creating a selection set to include the line drawn:

Command: **(entmake (list (cons 0 "line") '(10 0.0 0.0 0.0)'(11 0.0 10.0 0.0)))**
((0 . "line") (10 0.0 0.0 0.0) (11 0.0 10.0 0.0))

Command: **(setq gripset (ssget "_l"))**
<Selection set: a5>

Variable `gripset` points to the selection set created.

Draw side 2 and add it to the `gripset` selection set:

Command: **(entmake (list (cons 0 "line") '(10 0.0 10.0 0.0)'(11 10.0 10.0 0.0)))**
((0 . "line") (10 0.0 10.0 0.0) (11 10.0 10.0 0.0))

Command: **(ssadd (entlast) gripset)**
<Selection set: a5>

Create another selection set to include only side 2:

Command: **(setq 2onlyset (ssget "_l"))**
<Selection set: a8>

Draw side 3 and add it to the `gripset` selection set:

Command: **(entmake (list (cons 0 "line") '(10 10.0 10.0 0.0)'(11 10.0 0.0 0.0)))**
((0 . "line") (10 10.0 10.0 0.0) (11 10.0 0.0 0.0))

Command: **(ssadd (entlast) gripset)**
<Selection set: a5>

Create another selection and include side 3 in the selection set:

```
Command: (setq pickset (ssget "_l"))
<Selection set: ab>
```

Variable `pickset` points to the new selection set.

Draw side 4 and add it to the `gripset` and `pickset` selection sets:

```
Command: (entmake (list (cons 0 "line") '(10 10.0 0.0 0.0)'(11 0.0 0.0 0.0)))
((0 . "line") (10 10.0 0.0 0.0) (11 0.0 0.0 0.0))
```

```
Command: (ssadd (entlast) gripset)
<Selection set: a5>
```

```
Command: (ssadd (entlast) pickset)
<Selection set: ab>
```

At this point, `gripset` contains sides 1–4, `pickset` contains sides 3 and 4, and `2onlyset` contains only side 2.

Turn grip handles on for all objects in the `gripset` selection set:

```
Command: (sssetfirst gripset)
(<Selection set: a5>)
```

Turn grip handles off for all objects in `gripset`:

```
Command: (sssetfirst nil)
(nil)
```

Turn grip handles on and select all objects in `pickset`:

```
Command: (sssetfirst nil pickset)
(nil <Selection set: ab>)
```

Turn on grip handles for all objects in `2onlyset`, and select all objects in `pickset`:

```
Command: (sssetfirst 2onlyset pickset)
(<Selection set: a8> <Selection set: ab>)
```

Each **`sssetfirst`** call replaces the gripped and selected selection sets from the previous **`sssetfirst`** call. For example, after the following command is issued, grips are turned on in `2onlyset`, and no selection set is selected:

```
Command: (sssetfirst 2onlyset
(<Selection set: a8>)
```

NOTE Do *not* call `sssetfirst` when AutoCAD is in the middle of executing a command.

See Also

The ssget and ssgetfirst functions.

startapp

Starts a Windows application

(startapp *appcmd [file]*)

Arguments

appcmd	A string that specifies the application to execute. If *appcmd* does not include a full path name, `startapp` searches the directories in the PATH environment variable for the application.
file	A string that specifies the file name to be opened.

Return Values

An integer greater than 0, if successful, otherwise `nil`.

Examples

The following code starts the Windows Notepad and opens the *acad.lsp* file.

```
Command: (startapp "notepad" "acad.lsp")
33
```

If an argument has embedded spaces, it must be surrounded by literal double quotes. For example, to edit the file *my stuff.txt* with Notepad, use the following syntax:

```
Command: (startapp "notepad.exe" "\"my stuff.txt\"")
33
```

start_dialog

Displays a dialog box and begins accepting user input

```
(start_dialog)
```

You must first initialize the dialog box by a previous **`new_dialog`** call. The dialog box remains active until an action expression or callback function calls **`done_dialog`**. Usually **`done_dialog`** is associated with the tile whose `key` is `"accept"` (typically the OK button) and the tile whose `key` is `"cancel"` (typically the Cancel button).

The **`start_dialog`** function has no arguments.

Return Values

The **`start_dialog`** function returns the optional *status* passed to **`done_dialog`**. The default value is `1` if the user presses OK, `0` if the user presses Cancel, or `-1` if all dialog boxes are terminated with **`term_dialog`**. If **`done_dialog`** is passed an integer *status* greater than 1, **`start_dialog`** returns this value, whose meaning is determined by the application.

start_image

Starts the creation of an image in the dialog box tile

```
(start_image key)
```

Subsequent calls to **`fill_image`**, **`slide_image`**, and **`vector_image`** affect the created image until the application calls **`end_image`**.

Arguments

key	A string that specifies the dialog box tile. The *key* argument is case-sensitive.

Return Values

The *key* argument, if successful, `nil` otherwise.

NOTE Do not use the **`set_tile`** function between **`start_image`** and **`end_image`** function calls.

---

start_list

Starts the processing of a list in the list box or in the pop-up list dialog box tile

`(start_list` *`key [operation [index]]`*`)`

Subsequent calls to **`add_list`** affect the list started by **`start_list`** until the application calls **`end_list`**.

Arguments

key	A string that specifies the dialog box tile. The *key* argument is case-sensitive.
operation	An integer indicating the type of list operation to perform. You can specify one of the following: **1** Change selected list contents; **2** Append new list entry; **3** Delete old list and create new list (the default)
index	A number indicating the list item to change by the subsequent **`add_list`** call. The first item in the list is index 0. If not specified, *index* defaults to 0. The *index* argument is ignored if **`start_list`** is not performing a change operation.

Return Values

The name of the list that was started.

NOTE Do not use the **`set_tile`** function between **`start_list`** and **`end_list`** function calls.

strcase

Returns a string where all alphabetic characters have been converted to uppercase or lowercase

```
(strcase string [which])
```

Arguments

string	A string.
which	If specified as `T`, all alphabetic characters in *string* are converted to lowercase. Otherwise, characters are converted to uppercase.

Return Values

A string.

Examples

Command: **(strcase "Sample")**
"SAMPLE"

Command: **(strcase "Sample" T)**
"sample"

The **`strcase`** function will correctly handle case mapping of the currently configured character set.

strcat

Returns a string that is the concatenation of multiple strings

```
(strcat [string [string]...])
```

Arguments

string	A string.

Return Values

A string. If no arguments are supplied, **`strcat`** returns a zero-length string.

Examples

Command: **(strcat "a" "bout")**
"about"

Command: **(strcat "a" "b" "c")**
"abc"

Command: **(strcat "a" "" "c")**
"ac"

Command: **(strcat)**
""

strlen

Returns an integer that is the number of characters in a string

```
(strlen [string]...)
```

Arguments

string A string.

Return Values

An integer. If multiple *string* arguments are provided, **`strlen`** returns the sum of the lengths of all arguments. If you omit the arguments or enter an empty string, **`strlen`** returns 0.

Examples

```
Command (strlen "abcd")
4

Command (strlen "ab")
2

Command (strlen "one" "two" "four")
10

Command (strlen)
0

Command (strlen "")
0
```

subst

Searches a list for an old item and returns a copy of the list with a new item substituted in place of every occurrence of the old item

`(subst` *`newitem olditem lst`*`)`

Arguments

newitem	An atom or list.
olditem	An atom or list.
lst	A list.

Return Values

A list, with *newitem* replacing all occurrences of *olditem*. If *olditem* is not found in *lst*, **`subst`** returns *lst* unchanged.

Examples

Command: **(setq sample '(a b (c d) b))**
(A B (C D) B)

Command: **(subst 'qq 'b sample)**
(A QQ (C D) QQ)

Command: **(subst 'qq 'z sample)**
(A B (C D) B)

Command: **(subst 'qq '(c d) sample)**
(A B QQ B)

Command: **(subst '(qq rr) '(c d) sample)**
(A B (QQ RR) B)

Command: **(subst '(qq rr) 'z sample)**
(A B (C D) B)

When used in conjunction with **`assoc`**, **`subst`** provides a convenient means of replacing the value associated with one key in an association list, as demonstrated by the following function calls.

Set variable `who` to an association list:

Command: **(setq who '((first john) (mid q) (last public)))**
((FIRST JOHN) (MID Q) (LAST PUBLIC))

The following sets `old` to (FIRST JOHN) and `new` to (FIRST J):

Command: **(setq old (assoc 'first who) new '(first j))**
(FIRST J)

Finally, replace the value of the first item in the association list:

Command: **(subst new old who)**
((FIRST J) (MID Q) (LAST PUBLIC))

substr

Returns a substring of a string

`(substr` ***`string start [length]`*****`)`**

The **`substr`** function starts at the *start* character position of *string* and continues for *length* characters.

Arguments

string	A string.
start	A positive integer indicating the starting position in *string*. The first character in the string is position 1.
length	A positive integer specifying the number of characters to search through in *string*. If *length* is not specified, the substring continues to the end of *string*.

NOTE The first character of *string* is character number 1. This differs from other functions that process elements of a list (like **`nth`** and **`ssname`**) that count the first element as 0.

Return Values

A string.

Examples

Command: **(substr "abcde" 2)**
"bcde"

Command: **(substr "abcde" 2 1)**
"b"

Command: **(substr "abcde" 3 2)**
"cd"

tablet

Retrieves and sets digitizer (tablet) calibrations

`(tablet` ***`code [row1 row2 row3 direction]`*** **`)`**

Arguments

code	An integer that can be one of the following: **0** Return the current digitizer calibration. In this case, the remaining arguments must be omitted. **1** Set the calibration according to the arguments that follow. In this case, you must provide the new calibration settings (*row1, row2, row3,* and *direction*).
row1, row2, row3	Three 3D points. These three arguments specify the three rows of the tablet's transformation matrix. The third element in *row3* (*Z*) should always equal 1: **`tablet`** returns it as 1 even you specify a different value in *row3*.
direction	One 3D point. This is the vector (expressed in the World Coordinate System, or WCS) that is normal to the plane that represents the surface of the tablet. If the specified *direction* isn't normalized, **`tablet`** corrects it, so the *direction* it returns when you set the calibration may differ from the value you passed.

Return Values

If **`tablet`** fails, it returns `nil` and sets the `ERRNO` system variable to a value that indicates the reason for the failure (see appendix C, "AutoLISP Error Codes" in the *Visual LISP Developer's Guide*). This can happen if the digitizer is not a tablet.

Examples

A very simple transformation that can be established with **`tablet`** is the identity transformation:

```
(tablet 1 '(1 0 0) '(0 1 0) '(0 0 1) '(0 0 1))
```

With this transformation in effect, AutoCAD will receive, effectively, raw digitizer coordinates from the tablet. For example, if you pick the point with digitizer coordinates (5000,15000), AutoCAD will see it as the point in your drawing with those same coordinates.

The TABMODE system variable allows AutoLISP routines to toggle the tablet on and off.

See Also

The "Calibrating Tablets" topic in the *Visual LISP Developer's Guide*.

tblnext

Finds the next item in a symbol table

`(tblnext` ***`table-name [rewind]`*****`)`**

When **`tblnext`** is used repeatedly, it normally returns the next entry in the specified table each time. The **`tblsearch`** function can set the *next* entry to be retrieved. If the *rewind* argument is present and is not `nil`, the symbol table is rewound and the first entry in it is retrieved.

Arguments

table-name	A string that identifies a symbol table. Valid *table-name* values are `"LAYER"`, `"LTYPE"`, `"VIEW"`, `"STYLE"`, `"BLOCK"`, `"UCS"`, `"APPID"`, `"DIMSTYLE"`, and `"VPORT"`. The argument is not case sensitive.
rewind	If this argument is present and is not `nil`, the symbol table is rewound and the first entry in it is retrieved.

Return Values

If a symbol table entry is found, the entry is returned as a list of dotted pairs of DXF-type codes and values. If there are no more entries in the table, `nil` is returned. Deleted table entries are never returned.

Examples

Retrieve the first layer in the symbol table:

```
Command: (tblnext "layer" T)
((0 . "LAYER") (2 . "0") (70 . 0) (62 . 7) (6 . "CONTINUOUS"))
```

The return values represent the following:

`(0 . "LAYER")`	*Symbol type*
`(2 . "0")`	*Symbol name*
`(70 . 0)`	*Flags*
`(62 . 7)`	*Color number, negative if off*
`(6 . "CONTINUOUS")`	*Linetype name*

Note that there is no –1 group. AutoCAD remembers the last entry returned from each table and returns the next one each time **`tblnext`** is called for that table. When you begin scanning a table, be sure to supply a non-`nil` second argument to rewind the table and to return the first entry.

Entries retrieved from the block table include a –2 group with the entity name of the first entity in the block definition (if any). For example, the following command obtains information about a block called BOX:

```
Command: (tblnext "block")
((0 . "BLOCK") (2 . "BOX") (70 . 0) (10 9.0 2.0 0.0) (-2 . <Entity name: 1dca370>))
```

The return values represent the following:

`(0 . "BLOCK")`	*Symbol type*
`(2 . "BOX")`	*Symbol name*
`(70 . 0)`	*Flags*
`(10 9.0 2.0 0.0)`	*Origin X,Y,Z*
`(-2 . <Entity name: 1dca370>)`	*First entity*

The entity name in the –2 group is accepted by **`entget`** and **`entnext`**, but not by other entity access functions. For example, you cannot use **`ssadd`** to put it in a selection set. By providing the –2 group entity name to **`entnext`**, you can scan the entities comprising a block definition; **`entnext`** returns `nil` after the last entity in the block definition.

If a block contains no entities, the –2 group returned by **`tblnext`** is the entity name of its endblk entity.

NOTE The **`vports`** function returns current VPORT table information, therefore it may be easier to use **`vports`** as opposed to **`tblnext`** to retrieve this information.

---

tblobjname

Returns the entity name of a specified symbol table entry

`(tblobjname` *`table-name symbol`*`)`

Arguments

table-name	A string that identifies the symbol table to be searched. The argument is not case sensitive.
symbol	A string identifying the symbol to be searched for.

Return Values

The entity name of the symbol table entry, if found.

The entity name returned by **`tblobjname`** can be used in **`entget`** and **`entmod`** operations.

Examples

The following command searches for the entity name of the block entry "ESC-01":

Command: **(tblobjname "block" "ESC-01")**
<Entity name: 1dca368>

tblsearch

Searches a symbol table for a symbol name

`(tblsearch` *`table-name symbol [setnext]`*`)`

Arguments

table-name	A string that identifies the symbol table to be searched. This argument is not case sensitive.
symbol	A string identifying the symbol name to be searched for. This argument is not case sensitive.
setnext	If this argument is supplied and is not `nil`, the **`tblnext`** entry counter is adjusted so the following **`tblnext`** call returns the entry after the one returned by this **`tblsearch`**

call. Otherwise, **`tblsearch`** has no effect on the order of entries retrieved by **`tblnext`**.

Return Values

If **`tblsearch`** finds an entry for the given symbol name, it returns that entry in the format described for tblnext. If no entry is found, **`tblsearch`** returns `nil`.

Examples

The following command searches for a text style named “standard”:

Command: **(tblsearch "style" "standard")**
((0 . "STYLE") (2 . "STANDARD") (70 . 0) (40 . 0.0) (41 . 1.0) (50 . 0.0) (71 . 0) (42 . 0.3) (3 . "txt") (4 . ""))

term_dialog

Terminates all current dialog boxes as if the user had canceled each of them

```
(term_dialog)
```

If an application is terminated while any DCL files are open, AutoCAD automatically calls **`term_dialog`**. This function is used mainly for aborting nested dialog boxes.

Return Values

The **`term_dialog`** function always returns `nil`.

terpri

Prints a newline to the command line

```
(terpri)
```

The **`terpri`** function is not used for file I/O. To write a newline to a file, use **`prin1`**, **`princ`**, or **`print`**.

Return Values

`nil`

textbox

Measures a specified text object, and returns the diagonal coordinates of a box that encloses the text

(textbox *elist*)

Arguments

elist	An entity definition list defining a text object, in the format returned by **entget**.
	If fields that define text parameters other than the text itself are omitted from *elist*, the current (or default) settings are used.
	The minimum list accepted by **textbox** is that of the text itself.

Return Values

A list of two points, if successful, otherwise `nil`.

The points returned by **textbox** describe the bounding box of the text object as if its insertion point is located at (0,0,0) and its rotation angle is 0. The first list returned is generally the point (0.0 0.0 0.0) unless the text object is oblique or vertical, or it contains letters with descenders (such as *g* and *p*). The value of the first point list specifies the offset from the text insertion point to the lower-left corner of the smallest rectangle enclosing the text. The second point list specifies the upper-right corner of that box. Regardless of the orientation of the text being measured, the point list returned always describes the bottom-left and upper-right corners of this bounding box.

Examples

The following command supplies the text and accepts the current defaults for the remaining parameters:

```
Command: (textbox '((1 . "Hello world.")))
((0.000124126 -0.00823364 0.0) (3.03623 0.310345 0.0))
```

textpage

Switches from the graphics screen to the text screen

```
(textpage)
```

The **`textpage`** function is equivalent to **`textscr`**.

Return Values

`nil`

textscr

Switches from the graphics screen to the text screen (like the AutoCAD Flip Screen function key)

```
(textscr)
```

Return Values

The **`textscr`** function always returns `nil.`

See Also

The graphscr function.

trace

Aids in AutoLISP debugging

```
(trace [function...])
```

The **`trace`** function sets the trace flag for the specified functions. Each time a specified function is evaluated, a trace display appears showing the entry of the function (indented to the level of calling depth) and prints the result of the function.

If Visual LISP is active, trace output is sent to the Visual LISP Trace window. If Visual LISP is not active, trace output goes to the AutoCAD command window.

NOTE Once you start Visual LISP during an AutoCAD session, it remains active until you exit AutoCAD. Therefore, all **trace** output prints in the Visual LISP Trace window for the remainder of that AutoCAD session. Exiting or closing Visual LISP while AutoCAD is running only closes the IDE windows and places Visual LISP in a quiescent state; it does not result in a true shutdown. You must reopen Visual LISP to view the output in the Trace window.

Use **untrace** to turn off the trace flag.

Arguments

function	A symbol that names a function. If no argument is supplied, **trace** has no effect.

Return Values

The last function name passed to **trace**. If no argument is supplied, **trace** returns `nil`.

Examples

Define a function named **foo** and set the trace flag for the function:

```
Command: (defun foo (x) (if (> x 0) (foo (1- x))))
FOO
```

```
Command: (trace foo)
FOO
```

Invoke **foo** and observe the results:

```
Command: (foo 3)
Entering (FOO 3)
   Entering (FOO 2)
      Entering (FOO 1)
         Entering (FOO 0)
         Result:  nil
      Result:  nil
   Result:  nil
Result:  nil
```

Clear the trace flag by invoking **untrace**:

```
Command: (untrace foo)
FOO
```

See Also

The untrace function.

trans

Translates a point (or a displacement) from one coordinate system to another

(trans *pt from to [disp]*)

Arguments

pt	A list of three reals that can be interpreted as either a 3D point or a 3D displacement (vector).
from	An integer code, entity name, or 3D extrusion vector identifying the coordinate system in which *pt* is expressed. The integer code can be one of the following: **0** World (WCS) **1** User (current UCS) **2** If used with code 0 or 1, this indicates the Display Coordinate System (DCS) of the current viewport. When used with code 3, it indicates the DCS of the current model space viewport. **3** Paper space DCS (used *only* with code 2)
to	An integer code, entity name, or 3D extrusion vector identifying the coordinate system of the returned point. See the *from* argument for a list of valid integer codes.
disp	If present and is not `nil`, this argument specifies that *pt* is to be treated as a 3D displacement rather than as a point.

If you use an entity name for the *from* or *to* arguments, it must be passed in the format returned by the **`entnext`**, **`entlast`**, **`entsel`**, **`nentsel`**, and **`ssname`** functions. This format lets you translate a point to and from the Object Coordinate System (OCS) of a particular object. (For some objects, the OCS is equivalent to the WCS; for these objects, conversion between OCS and WCS is a null operation.) A 3D extrusion vector (a list of three reals) is another method of converting to and from an object's OCS. However, this does not work for those objects whose OCS is equivalent to the WCS.

Return Values

A 3D point (or displacement) in the requested *to* coordinate system.

Examples

In the following examples, the UCS is rotated 90 degrees counterclockwise around the World *Z* axis:

```
Command: (trans '(1.0 2.0 3.0) 0 1)
(2.0 -1.0 3.0)
```

```
Command: (trans '(1.0 2.0 3.0) 1 0)
(-2.0 1.0 3.0)
```

The coordinate systems are discussed in greater detail in the *Visual LISP Developer's Guide*, under the topic, "Coordinate System Transformations."

For example, to draw a line from the insertion point of a piece of text (without using Osnap), you convert the text object's insertion point from the text object's OCS to the UCS.

```
(trans text-insert-point text-ename 1)
```

You can then pass the result to the From point prompt.

Conversely, you must convert point (or displacement) values to their destination OCS before feeding them to **entmod**. For example, if you want to move a circle (without using the MOVE command) by the UCS-relative offset (1,2,3), you need to convert the displacement from the UCS to the circle's OCS:

```
(trans '(1 2 3) 1 circle-ename)
```

Then you add the resulting displacement to the circle's center point.

For example, if you have a point entered by the user and want to find out which end of a line it looks closer to, you convert the user's point from the UCS to the DCS.

```
(trans user-point 1 2)
```

Then you convert each of the line's endpoints from the OCS to the DCS.

```
(trans endpoint line-ename 2)
```

From there you can compute the distance between the user's point and each endpoint of the line (ignoring the *Z* coordinates) to determine which end looks closer.

The **trans** function can also transform 2D points. It does this by setting the *Z* coordinate to an appropriate value. The *Z* component used depends on the *from* coordinate system that was specified and on whether the value is to be converted as a point or as a displacement. If the value is to be converted as a displacement, the *Z* value is always 0.0; if the value is to be converted as a point, the filled-in *Z* value is determined as shown in the following table.

Converted 2D point *Z* values

From	Filled-in *Z* value
WCS	0.0
UCS	Current elevation
OCS	0.0
DCS	Projected to the current construction plane (UCS *XY* plane + current elevation)
PSDCS	Projected to the current construction plane (UCS *XY* plane + current elevation)

type

Returns the type of a specified item

```
(type item)
```

Arguments

item	A symbol.

Return Values

The data type of *item*. Items that evaluate to `nil` (such as unassigned symbols) return `nil`. The data type is returned as one of the atoms listed in the following table:

Data types returned by the type function

Data type	Description
ENAME	Entity names
EXRXSUBR	External ObjectARX applications

Data types returned by the type function (*continued*)

Data type	Description
FILE	File descriptors
INT	Integers
LIST	Lists
PAGETB	Function paging table
PICKSET	Selection sets
REAL	Floating-point numbers
SAFEARRAY	Safearray
STR	Strings
SUBR	Internal AutoLISP functions or functions loaded from compiled (FAS or VLX) files. Functions in LISP source files loaded from the AutoCAD Command prompt may also appear as SUBR.
SYM	Symbols
VARIANT	Variant
USUBR	User-defined functions loaded from LISP source files
VLA-object	ActiveX objects

Examples

For example, given the following assignments:

```
(setq a 123 r 3.45 s "Hello!" x '(a b c))
(setq f (open "name" "r"))
```

then

```
(type 'a)                    returns  SYM
(type a)                     returns  INT
(type f)                     returns  FILE
(type r)                     returns  REAL
(type s)                     returns  STR
(type x)                     returns  LIST
(type +)                     returns  SUBR
(type nil)                   returns  nil
```

The following code example uses the **type** function on the argument passed to it:

```
(defun isint (a)
   (if (= (type a) 'INT)      is TYPE integer?
     T                        yes, return T
     nil                      no, return nil
   )
)
```

unload_dialog

Unloads a DCL file

(unload_dialog *dcl_id*)

Unloads the DCL file associated with *dcl_id* (obtained from a previous **new_dialog** call) from memory.

It is generally not necessary to unload a DCL definition from memory, unless you are running low on memory or need to update the DCL dialog definition from a new file.

Arguments

dcl_id	A DCL file identifier obtained from a previous **load_dialog** call.

Return Values

The **unload_dialog** function always returns `nil`.

See Also

The load_dialog and new_dialog functions.

untrace

Clears the trace flag for the specified functions

(untrace *[function...]*)

Arguments

function	A symbol that names a function. If *function* is not specified, **untrace** has no effect.

Return Values

The last function name passed to **`untrace`**. If *function* was not specified, **`untrace`** returns `nil`.

Examples

The following command clears the trace flag for function **`foo`**:

```
Command: (untrace foo)
FOO
```

See Also

The trace function.

vector_image

Draws a vector in the currently active dialog box image

`(vector_image` ***`x1 y1 x2 y2 color`*****`)`**

This function draws a vector in the currently active dialog box image (opened by **`start_image`**) from the point (*x1*,*y1*) to (*x2*,*y2*). The origin (0,0) is the upper-left corner of the image. You can obtain the coordinates of the lower-right corner by calling the dimension functions (**`dimx_tile`** and **`dimy_tile`**).

Arguments

x1	*X* coordinate of the first point.
y1	*Y* coordinate of the first point.
x2	*X* coordinate of the second point.
y2	*Y* coordinate of the second point.
color	An AutoCAD color number, or one of the logical color numbers shown in the following table:

Symbolic names for color attribute

Color number	ADI mnemonic	Description
–2	BGLCOLOR	Current background of the AutoCAD graphics screen
–15	DBGLCOLOR	Current dialog box background color
–16	DFGLCOLOR	Current dialog box foreground color (text)
–18	LINELCOLOR	Current dialog box line color

Return Values

An integer representing the color of the vector.

Examples

```
(setq color -2) ;; color of AutoCAD background screen
(vector_image
  0
  0
  (dimx_tile "slide_tile")
  (dimy_tile "slide_tile")
  color
)
(end_image)
```

ver

Returns a string that contains the current AutoLISP version number

`(ver)`

The **`ver`** function can be used to check the compatibility of programs.

Return Values

The string returned takes the following form:

`"Visual LISP` *`version`* `(`*`nn`*`)"`

where *version* is the current version number and *nn* is a two-letter language description.

Examples of the two-letter language descriptions are as follows:

(de) German
(en) US/UK
(es) Spanish
(fr) French
(it) Italian

Examples

Command: **(ver)**
"Visual LISP 2000 (en)"

vl-acad-defun

Defines an AutoLISP function symbol as an external subroutine

(vl-acad-defun '*symbol*)

symbol	A symbol identifying a function.

If a function does not have the `c:` prefix, and you want to be able to invoke this function from an external ObjectARX application, you can use **`vl-acad-defun`** to make the function accessible.

Return Values

Unspecified.

vl-acad-undefun

Undefines an AutoLISP function symbol so it is no longer available to ObjectARX applications

(vl-acad-undefun '*symbol*)

symbol	A symbol identifying a function.

You can use **`vl-acad-undefun`** to undefine a `c:` function or a function that was exposed via **`vl-acad-defun`**.

Return Values

`T`, if successful, `nil`, if unsuccessful (for example, the function was not defined in AutoLISP).

vl-arx-import

Imports ObjectARX/ADSRX functions into a separate-namespace VLX

(vl-arx-import *['function* | "*application*"*]*)

By default, separate-namespace VLX applications do not import any functions from ObjectARX/ADSRX applications. Use **vl-arx-import** to explicitly import functions from ObjectARX/ADSRX applications.

Arguments

function	A symbol naming the function to import.
application	A string naming the application whose functions are to be imported.

If no argument (or nil) is specified, **vl-arx-import** imports all function names from the current document namespace.

Return Values

Unspecified.

If executed from a document VLX, this function does nothing and returns nil, as all ADS-DEFUN function names are automatically imported to document VLX applications.

Examples

To see how **vl-arx-import** works, try the following:

1 Copy the following code into the VLISP editor and save the file:

```
(vl-doc-export 'testarx)
(defun testarx ()
   (princ "This function tests invoking an ARX app ")
   (vl-arx-import 'c:cal)
   (c:cal)
)
```

2 Use Make Application to build a VLX with this code. Select Separate-Namespace Application Options.

3 Load *geomcal.arx*, if it is not already loaded.

4 Load and run the application.

To verify the effect of **vl-arx-import**, comment out the **vl-arx-import** call in the code, save the change, then rebuild and run the application. Without the **vl-arx-import** call, the **c:cal** function will not be found.

In the example above, you could have replaced the **`vl-arx-import`** call with the following:

```
(vl-arx-import "geomcal.arx")
```

This would import all functions defined in *geomcal.arx*, including **`c:cal`**.

vl-bb-ref

Returns the value of a variable from the blackboard namespace

```
(vl-bb-ref 'variable)
```

Arguments

'variable	A symbol identifying the variable to be retrieved.

Return Values

The value of the variable named by symbol.

Examples

Set a variable in the blackboard:

Command: **(vl-bb-set** '**foobar** "**Root toot toot**"**)**
"Root toot toot"

Use **`vl-bb-ref`** to retrieve the value of `foobar` from the blackboard:

Command: **(vl-bb-ref** '**foobar)**
"Root toot toot"

See Also

The vl-bb-set function. Also, see "Sharing Data between Namespaces" in the *Visual LISP Developer's Guide* for a description of the blackboard namespace.

vl-bb-set

Sets a variable in the blackboard namespace

```
(vl-bb-set 'symbol value)
```

Arguments

'symbol	A symbol naming the variable to be set.
value	Any value, except a function.

Return Values

The *value* you assigned to *symbol*.

Examples

Command: **(vl-bb-set 'foobar "Root toot toot")**
"Root toot toot"

Command: **(vl-bb-ref 'foobar)**
"Root toot toot"

See Also

The vl-bb-ref function. Also, see “Sharing Data between Namespaces” in the *Visual LISP Developer’s Guide* for a description of the blackboard namespace.

vl-catch-all-apply

Passes a list of arguments to a specified function and traps any exceptions

```
(vl-catch-all-apply 'function list)
```

Arguments

'function	A function. The *function* argument can be either a symbol identifying a `defun`, or a `lambda` expression.
list	A list containing arguments to be passed to the function.

Return Values

The result of the function call, if successful. If an error occurs, **vl-catch-all-apply** returns an error object.

Examples

If the function invoked by **vl-catch-all-apply** completes successfully, it is the same as using **apply**, as the following examples show:

```
_$ (setq catchit (apply '/ '(50 5)))
10
```

```
_$ (setq catchit (vl-catch-all-apply '/ '(50 5)))
10
```

The benefit of using **vl-catch-all-apply** is that it allows you to intercept errors and continue processing. Look at what happens when you try to divide by zero using **apply**:

```
_$ (setq catchit (apply '/ '(50 0)))
; error: divide by zero
```

When you use **apply**, an exception occurs and an error message displays.

Here is the same operation using **vl-catch-all-apply**

```
_$ (setq catchit (vl-catch-all-apply '/ '(50 0)))
#<%catch-all-apply-error%>
```

The **vl-catch-all-apply** function traps the error and returns an error object. Use **vl-catch-all-error-message** to see the error message contained in the error object:

```
_$ (vl-catch-all-error-message catchit)
"divide by zero"
```

See Also

The vl-catch-all-error-message and vl-catch-all-error-p functions in this reference and “Error Handling” in the *Visual LISP Developer’s Guide*.

vl-catch-all-error-message

Returns a string from an error object

(vl-catch-all-error-message *error-obj*)

Arguments

error-obj	An error object returned by **vl-catch-all-apply**.

Return Values

A string containing an error message.

Examples

Divide by zero using **`vl-catch-all-apply`**:

```
_$ (setq catchit (vl-catch-all-apply '/ '(50 0)))
#<%catch-all-apply-error%>
```

The **`vl-catch-all-apply`** function traps the error and returns an error object. Use **`vl-catch-all-error-message`** to see the error message contained in the error object:

```
_$ (vl-catch-all-error-message catchit)
"divide by zero"
```

See Also

The vl-catch-all-apply and vl-catch-all-error-p functions in this reference and "Error Handling" in the *Visual LISP Developer's Guide*.

vl-catch-all-error-p

Determines whether an argument is an error object returned from `vl-catch-all-apply`

(vl-catch-all-error-p *arg*)

Arguments

arg	Any argument.

Return Values

`T`, if the supplied argument is an error object returned from **`vl-catch-all-apply`**, `nil` otherwise.

Examples

Divide by zero using **`vl-catch-all-apply`**:

```
_$ (setq catchit (vl-catch-all-apply '/ '(50 0)))
#<%catch-all-apply-error%>
```

Use **`vl-catch-all-error-p`** to determine if the value returned by **`vl-catch-all-apply`** is an error object:

```
_$ (vl-catch-all-error-p catchit)
T
```

See Also

The vl-catch-all-apply and vl-catch-all-error-message functions, and “Error Handling” in the *Visual LISP Developer’s Guide*.

vl-cmdf

Executes an AutoCAD command

Arguments

`(vl-cmdf [arguments] ...)`

The **`vl-cmdf`** function is similar to the **`command`** function, but differs from **`command`** in the way it evaluates the arguments passed to it. The **`vl-cmdf`** function evaluates all the supplied arguments before executing the AutoCAD command, and will not execute the AutoCAD command if it detects an error during argument evaluation. In contrast, the **`command`** function passes each argument in turn to AutoCAD, so the command may be partially executed before an error is detected.

If your command call includes a call to another function, **`vl-cmdf`** executes the call *before* it executes your command, while **`command`** executes the call *after* it begins executing your command.

Some AutoCAD commands may work correctly when invoked through **`vl-cmdf`**, while failing when invoked through **`command`**. The **`vl-cmdf`** function mainly overcomes the limitation of not being able to use `get`*xxx* functions inside **`command`**.

Arguments

arguments	AutoCAD commands and their options. The *arguments* to the **`vl-cmdf`** function can be strings, reals, integers, or points, as expected by the prompt sequence of the executed command. A null string (`""`) is equivalent to pressing ENTER on the keyboard. Invoking **`vl-cmdf`** with no argument is equivalent to pressing ESC and cancels most AutoCAD commands.

Return Values

`T`

Note that if you issue **`vl-cmdf`** from Visual LISP, focus does not change to the AutoCAD window. If the command requires user input, you'll see the return value (`T`) in the Console window, but AutoCAD will be waiting for input. You must manually activate the AutoCAD window and respond to the prompts. Until you do so, any subsequent commands will fail.

Examples

The differences between **`command`** and **`vl-cmdf`** are easier to see if you enter the following calls at the AutoCAD Command prompt, rather than the VLISP Console prompt:

```
Command: (command "line" (getpoint "point?") '(0 0) "")
line Specify first point: point?
Specify next point or [Undo]:
Command: nil
```

Using **`command`**, the LINE command executes first, then the **`getpoint`** function is called.

```
Command: (VL-CMDF "line" (getpoint "point?") '(0 0) "")
point?line Specify first point:
Specify next point or [Undo]:
Command: T
```

Using **`vl-cmdf`**, the **`getpoint`** function is called first (notice the "point?" prompt from **`getpoint`**), then the LINE command executes.

The following examples show the same commands, but pass an invalid point list argument to the LINE command. Notice how the results differ:

```
Command: (command "line" (getpoint "point?") '(0) "")
line Specify first point: point?
Specify next point or [Undo]:
Command: ERASE nil
Select objects: Specify opposite corner: *Cancel*
0 found
```

The **`command`** function passes each argument in turn to AutoCAD, without evaluating the argument, so the invalid point list is undetected.

```
Command: (VL-CMDF "line" (getpoint "point?") '(0) "")
point?Application ERROR: Invalid entity/point list.
nil
```

Because **`vl-cmdf`** evaluates each argument before passing the command to AutoCAD, the invalid point list is detected and the command is not executed.

See Also

The command function.

vl-consp

Determines whether or not a list is `nil`

```
(vl-consp list-variable)
```

The **`vl-consp`** function determines whether a variable contains a valid list definition.

Arguments

list-variable	A list.

Return Values

`T`, if *list-variable* is a list and is not `nil`, otherwise `nil`.

Examples

```
_$ (vl-consp nil)
nil

_$ (vl-consp t)
nil

_$ (vl-consp (cons 0 "LINE"))
T
```

vl-directory-files

Lists all files in a given directory

```
(vl-directory-files [directory pattern directories])
```

Arguments

directory	A string naming the directory to collect files for; if `nil` or absent, **`vl-directory-files`** uses the current directory.

pattern	A string containing a DOS pattern for the file name; if `nil` or absent, **`vl-directory-files`** assumes "*.*"
directories	An integer that indicates whether the returned list should include directory names. Specify one of the following:

–1 List directories only.

0 List files and directories (the default).

1 List files only.

Return Values

A list of file and path names, or `nil`, if no files match the specified pattern.

Examples

```
_$ (vl-directory-files "c:/acadwin" "acad*.exe")
("ACAD.EXE" "ACADAPP.EXE" "ACADL.EXE" "ACADPS.EXE")
```

```
_$ (vl-directory-files "e:/acadwin" nil -1)
("." ".." "SUPPORT" "SAMPLE" "ADS" "FONTS" "IGESFONT" "SOURCE"
"ASE")
```

```
_$ (vl-directory-files "E:/acad13c4" nil -1)
("." ".." "WIN" "COM" "DOS")
```

vl-doc-export

Makes a function available to the current document

`(vl-doc-export 'function)`

When issued from a VLX that runs in its own namespace, **`vl-doc-export`** exposes the specified function to any document namespace that loads the VLX.

The **`vl-doc-export`** function should only be used at the top-level in a file, never inside other forms (for example, not within a **`defun`**).

Arguments

'function	A symbol naming the function to be exported.

Return Values

Unspecified.

Examples

The following code shows the contents of a file named *kertrats.lsp*. This file is compiled into a VLX that runs in its own namespace. The VLX file is named *kertrats.vlx*. The **`vl-doc-export`** call makes the **`kertrats`** function visible to any document that loads *kertrats.vlx*:

```
(vl-doc-export 'kertrats)
(defun kertrats ()
  (princ "This function goes nowhere")
)
```

vl-doc-import

Imports a previously exported function into a VLX namespace

`(vl-doc-import` ***`application ['function...]`*** **`)`**

This function can be used in a separate-namespace VLX to import a function that was previously exported from another VLX loaded from the same document.

The **`vl-doc-import`** function should only be used at the top-level in a file, never inside other forms (for example, not within a **`defun`**).

Arguments

application	A string naming the VLX application whose functions are to be imported. Do not include the *.vlx* extension in the name.
function	One or more symbols naming functions to be imported. If no functions are specified, all functions exported by *application* will be imported.

Return Values

Unspecified.

Examples

Import function **`ldataget`** from the **`ldatatest`** application:

```
(vl-doc-import "ldatatest" 'ldataget)
nil
```

vl-doc-ref

Retrieves the value of a variable from the current document’s namespace.

This function can be used by a separate-namespace VLX application to retrieve the value of a variable from the current document’s namespace.

```
(vl-doc-ref 'symbol)
```

Arguments

’symbol	A symbol naming a variable.

Return Values

The value of the variable identified by *symbol*.

Examples

Command: **(vl-doc-ref 'foobar)**
"Rinky dinky stinky"

See Also

The vl-doc-set function.

vl-doc-set

Sets the value of a variable in the current document’s namespace.

```
(vl-doc-set 'symbol value)
```

This function can be used by a VLX application to set the value of a variable that resides in the current document’s namespace.

If executed within a document namespace, `vl-doc-set` is equivalent to `set`.

Arguments

’symbol	A symbol naming a variable.
value	Any value.

Return Values

The *value* set.

Examples

Command: **(vl-doc-set 'foobar "Rinky dinky stinky")**
"Rinky dinky stinky"

See Also

The vl-doc-ref function.

vl-every

Checks whether the predicate is true for every element combination

(vl-every *predicate-function list [list]...*)

The **vl-every** function passes the first element of each supplied list as an argument to the test function, followed by the next element from each list, and so on. Evaluation stops as soon as one of the lists runs out.

Arguments

predicate-function	The test function. This can be any function that accepts as many arguments as there are lists provided with **vl-every**, and returns T on any user-specified condition. The *predicate-function* value can take one of the following forms: ■ *A symbol (function name)* ■ '(LAMBDA (A1 A2) ...) ■ (FUNCTION (LAMBDA (A1 A2) ...))
list	A list to be tested.

Return Values

T, if *predicate-function* returns a non-nil value for every element combination, nil otherwise.

Examples

Check whether there are any empty files in the current directory:

```
_$ (vl-every
'(lambda (fnm) (> (vl-file-size fnm) 0))
   (vl-directory-files nil nil 1) )
T
```

Check whether the list of numbers in NLST is ordered by '<=:

```
_$ (setq nlst (list 0 2 pi pi 4))
(0 2 3.14159 3.14159 4)

_$ (vl-every '<= nlst (cdr nlst))
T
```

Compare the results of the following expressions:

```
_$ (vl-every '= '(1 2) '(1 3))
nil

_$ (vl-every '= '(1 2) '(1 2 3))
T
```

The first expression returned `nil` because **`vl-every`** compared the second element in each list and they were not numerically equal. The second expression returned `T` because **`vl-every`** stopped comparing elements after it had processed all the elements in the shorter list (1 2), at which point the lists were numerically equal. If the end of a list is reached, **`vl-every`** returns a non-`nil` value.

The following example demonstrates the result when **`vl-every`** evaluates one list that contains integer elements and another list that is `nil`:

```
_$ (setq alist (list 1 2 3 4))
(1 2 3 4)

_$ (setq junk nil)
nil

_$ (vl-every '= junk alist)
T
```

The return value is `T` because **`vl-every`** responds to the `nil` list as if it has reached the end of the list (even though the predicate hasn't yet been applied to any elements). And since the end of a list has been reached, **`vl-every`** returns a non-`nil` value.

vl-exit-with-error

Passes control from a VLX error handler to the `*error*` function of the calling namespace

`(vl-exit-with-error` ***`msg`*****`)`**

This function is used by VLX applications that run in their own namespace. When **`vl-exit-with-error`** executes, it calls the **`*error*`** function, the stack is unwound, and control returns to a command prompt.

Arguments

msg	A string.

Return Values

None.

Examples

The following code illustrates the use of **`vl-exit-with-error`** to pass a string to the **`*error*`** function of the calling namespace:

```
(defun *error* (msg)
  ... ; processing in VLX namespace/execution context
(vl-exit-with-error (strcat "My application bombed! " msg)))
```

See Also

The vl-exit-with-value function and "Handling Errors in an MDI Environment" in the *Visual LISP Developer's Guide*.

vl-exit-with-value

Returns a value to the function that invoked the VLX from another namespace

`(vl-exit-with-value `*`value`*`)`

A VLX **`*error*`** handler can use the **`vl-exit-with-value`** function to return a value to the program that called the VLX.

Arguments

value	Any value.

Return Values

value

Examples

The following example uses **`vl-exit-with-value`** to return the integer value 3 to the function that invoked the VLX:

```
(defun *error* (msg)
  ... ; processing in VLX-T namespace/execution context
  (vl-exit-with-value  3))
```

See Also

The vl-exit-with-error function and the “Handling Errors in an MDI Environment” topic in the *Visual LISP Developer’s Guide*.

vl-file-copy

Copies or appends the contents of one file to another file

(vl-file-copy *source-file* *destination-file* *[append]*)

Copy or append the contents of one file to another file. The **`vl-file-copy`** function will not overwrite an existing file, only append to it.

Arguments

source-file	A string naming the file to be copied. If you do not specify a full path name, **`vl-file-copy`** looks in the AutoCAD start-up directory.
destination-file	A string naming the destination file. If you do not specify a path name, **`vl-file-copy`** writes to the AutoCAD start-up directory.
append	If specified and not `nil`, *source-file* is appended to *destination-file* (that is, copied to the end of the destination file).

Return Values

An integer, if the copy was successful, otherwise `nil`.

Some typical reasons for returning `nil` are:

- *source-file* is not readable
- *source-file* is a directory
- *append?* is absent or `nil` and *destination-file* exists
- *destination-file* cannot be opened for output (that is, it is an illegal file name or a write-protected file)
- *source-file* is the same as *destination-file*

Examples

Copy *autoexec.bat* to *newauto.bat*:

```
_$ (vl-file-copy "c:/autoexec.bat" "c:/newauto.bat")
1417
```

Copy *test.bat* to *newauto.bat*:

```
_$ (vl-file-copy "c:/test.bat" "c:/newauto.bat")
nil
```

The copy fails because *newauto.bat* already exists, and the *append* argument was not specified.

Repeat the previous command, but specify *append*:

```
_$ (vl-file-copy "c:/test.bat" "c:/newauto.bat" T)
185
```

The copy is successful because `T` was specified for the *append* argument.

vl-file-delete

Deletes a file

```
(vl-file-delete filename)
```

Arguments

filename	A string containing the name of the file to be deleted. If you do not specify a full path name, **`vl-file-delete`** searches the AutoCAD start-up directory.

Return Values

`T`, if successful, `nil` if delete failed.

Examples

Delete *newauto.bat*:

```
_$ (vl-file-delete "newauto.bat")
nil
```

Nothing was deleted because there is no *newauto.bat* file in the AutoCAD start-up directory.

Delete the *newauto.bat* file in the *c:\* directory:

```
_$ (vl-file-delete "c:/newauto.bat")
T
```

The delete was successful because the full path name identified an existing file.

vl-file-directory-p

Determines if a file name refers to a directory

(vl-file-directory-p *filename*)

Arguments

filename	A string containing a file name. If you do not specify a full path name, **vl-file-directory-p** searches only the AutoCAD start-up directory.

Return Values

`T`, if *filename* is the name of a directory, `nil` if it is not.

Examples

```
_$ (vl-file-directory-p "sample")
T

_$ (vl-file-directory-p "yinyang")
nil

_$ (vl-file-directory-p "c:/program files/autocad 2000i")
T

_$ (vl-file-directory-p "c:/program files/autocad 2000i/visuallisp/
yinyang.lsp")
nil
```

vl-file-rename

Renames a file

(vl-file-rename *old-filename new-filename*)

Arguments

old-filename	A string containing the name of the file you want to rename. If you do not specify a full path name, **vl-file-rename** looks in the AutoCAD start-up directory.
new-filename	A string containing the new name to be assigned to the file.

NOTE If you do not specify a path name, **vl-file-rename** writes the renamed file to the AutoCAD start-up directory.

Return Values

`T`, if renaming completed successfully, `nil` if renaming failed.

Examples

```
_$ (vl-file-rename "c:/newauto.bat" "c:/myauto.bat")
T
```

vl-file-size

Determines the size of a file, in bytes

`(vl-file-size` ***`filename`*****`)`**

Arguments

filename	A string naming the file to be sized. If you do not specify a full path name, **`vl-file-size`** searches the AutoCAD start-up directory for the file.

Return Values

If successful, **`vl-file-size`** returns an integer showing the size of *filename*. If the file is not readable, **`vl-file-size`** returns `nil`. If *filename* is a directory or an empty file, **`vl-file-size`** returns 0.

Examples

```
_$ (vl-file-size "c:/autoexec.bat")
1417
```

```
_$ (vl-file-size "c:/")
0
```

In the preceding example, **`vl-file-size`** returned 0 because *c:/* names a directory.

vl-file-systime

Returns last modification time of the specified file

`(vl-file-systime` ***`filename`*****`)`**

Arguments

filename	A string containing the name of the file to be checked.

Return Values

A list containing the modification date and time, or `nil`, if the file is not found.

The list returned contains the following elements:

- year
- month
- day-of-week
- day-of-month
- hours
- minutes
- seconds

Note that Monday is day 1 of day-of-week, Tuesday is day 2, etc.

Examples

```
_$ (vl-file-systime
   "c:/program files/autocad 2000i/sample/visuallisp/yinyang.lsp")
(1998 4 3 8 10 6 52 0)
```

The returned value shows that the file was last modified in 1998, in the 4th month of the year (April), the 3rd day of the week (Wednesday), on the 10th day of the month, at 6:52.

vl-filename-base

Returns the name of a file, after stripping out the directory path and extension

```
(vl-filename-base filename)
```

Arguments

filename	A string containing a file name. The **`vl-filename-base`** function does not check to see if the file exists.

Return Values

A string containing *filename* in uppercase, with any directory and extension stripped from the name.

Examples

```
_$ (vl-filename-base "c:\\acadwin\\acad.exe")
"ACAD"

_$ (vl-filename-base "c:\\acadwin")
"ACADWIN"
```

vl-filename-directory

Returns the directory path of a file, after stripping out the name and extension

```
(vl-filename-directory filename)
```

Arguments

filename	A string containing a complete file name, including the path. The **`vl-filename-directory`** function does not check to see if the specified file exists. Slashes (/) and backslashes (\) are accepted as directory delimiters.

Return Values

A string containing the directory portion of *filename*, in uppercase.

Examples

```
_$ (vl-filename-directory "c:\\acadwin\\acad.exe")
"C:\\ACADWIN"

_$ (vl-filename-directory "acad.exe")
""
```

vl-filename-extension

Returns the extension from a file name, after stripping out the rest of the name

```
(vl-filename-extension filename)
```

Arguments

filename	A string containing a file name, including the extension. The **`vl-filename-extension`** function does not check to see if the specified file exists.

Return Values

A string containing the extension of *filename*. The returned string starts with a period (.) and is in uppercase. If *filename* does not contain an extension, **`vl-filename-extension`** returns `nil`.

Examples

```
_$ (vl-filename-extension "c:\\acadwin\\acad.exe")
".EXE"

_$ (vl-filename-extension "c:\\acadwin\\acad")
nil
```

vl-filename-mktemp

Calculates a unique file name to be used for a temporary file

`(vl-filename-mktemp` ***`[pattern directory extension]`*** **`)`**

Arguments

pattern	A string containing a file name pattern; if `nil` or absent, **`vl-filename-mktemp`** uses "$VL~~".
directory	A string naming the directory for temporary files; if `nil` or absent, **`vl-filename-mktemp`** chooses a directory in the following order: ■ *The directory specified in pattern, if any.* ■ *The directory specified in the* `TMP` *environment variable.* ■ *The directory specified in the* `TEMP` *environment variable.* ■ The current directory.
extension	A string naming the extension to be assigned to the file; if `nil` or absent, **`vl-filename-mktemp`** uses the extension part of *pattern* (which may be an empty string).

Return Values

A string containing a file name, in the following format:

`directory``\`*`base`*`<`*`XXX`*`><.`*`extension`*`>`

where:

base is up to 5 characters, taken from *pattern*

XXX is a 3 character unique combination

All file names generated by **`vl-filename-mktemp`** during a VLISP session are deleted when you exit VLISP.

Examples

```
_$ (vl-filename-mktemp)
"C:\\TMP\\$VL~~004"

_$ (vl-filename-mktemp "myapp.del")
"C:\\TMP\\MYAPP005.DEL"

_$ (vl-filename-mktemp "c:\\acadwin\\myapp.del")
"C:\\ACADWIN\\MYAPP006.DEL"

_$ (vl-filename-mktemp "c:\\acadwin\\myapp.del")
"C:\\ACADWIN\\MYAPP007.DEL"

_$ (vl-filename-mktemp "myapp" "c:\\acadwin")
"C:\\ACADWIN\\MYAPP008"

_$ (vl-filename-mktemp "myapp" "c:\\acadwin" ".del")
"C:\\ACADWIN\\MYAPP00A.DEL"
```

vl-get-resource

Returns the text stored in a *.txt* file packaged in a VLX

`(vl-get-resource` ***`text-file`*****`)`**

Arguments

text-file	A string naming a *.txt* file packaged with the VLX. Do not include the *.txt* extension when specifying the file name.

Return Values

A string containing the text in *text-file*.

Examples

Assume the *getres.vlx* file contains a LISP program defining a function named **`print-readme`**, and a text file named *readme.txt*. The **`print-readme`** function is defined as follows:

```
(defun print-readme ()
   (princ (vl-get-resource "readme"))
   (princ)
  )
```

After loading *getres.vlx*, invoke **`print-readme`**:

```
_$ (print-readme)
There is very important information here!
Be sure to thoroughly read the following!
Are you ready?
Here it comes...
```

vl-list*

Constructs and returns a list

`(vl-list* object [object]...)`

Arguments

object	Any LISP object.

Return Values

The **`vl-list*`** function is similar to **`list`**, but it will place the last *object* in the final **`cdr`** of the result list. If the last argument to **`vl-list*`** is an atom, the result is a dotted list. If the last argument is a list, its elements are appended to all previous arguments added to the constructed list. The possible return values from **`vl-list*`** are:

- An atom, if a single atom *object* is specified.
- A dotted pair, if all *object* arguments are atoms.
- A dotted list, if the last argument is an atom and neither of the previous conditions are true.
- A list, if none of the previous statements are true.

Examples

```
_$ (vl-list* 1)
1

_$ (vl-list* 0 "text")
(0 . "TEXT")

_$ (vl-list* 1 2 3)
(1 2 . 3)

_$ (vl-list* 1 2 '(3 4))
(1 2 3 4)
```

See Also

The list function.

vl-list->string

Combines the characters associated with a list of integers into a string

```
(vl-list->string char-codes-list)
```

Arguments

char-codes-list	A list of non-negative integers. Each integer must be less than 256.

Return Values

A string of characters, with each character based on one of the integers supplied in *char-codes-list*.

Examples

```
_$ (vl-list->string nil)
""

_$ (vl-list->string '(49 50))
"12"
```

See Also

The vl-string->list function.

vl-list-exported-functions

Lists exported functions

```
(vl-list-exported-functions [appname])
```

Arguments

appname	A string naming a loaded VLX application. Do *not* include the *.vlx* extension.

Return Values

A list of strings naming exported functions, or `nil`, if there are no functions exported from the specified VLX. If *appname* is omitted or is `nil`, **`vl-list-exported-functions`** returns a list of all exported functions (for example, *c:* functions) except those exported from VLX namespaces.

Examples

```
_$ (vl-list-exported-functions "whichexpns")
("WHICHNAMESPACE")
```

See Also

The vl-list-loaded-vlx function.

vl-list-length

Calculates list length of a true list

(vl-list-length *list-or-cons-object*)

Arguments

list-or-cons-object	A true or dotted list.

Return Values

An integer containing the list length, if the argument is a true list, or `nil`, if *list-or-cons-object* is a dotted list.

Compatibility note: The **`vl-list-length`** function returns `nil` for a dotted list, while the corresponding Common Lisp function issues an error message if the argument is a dotted list.

Examples

```
_$ (vl-list-length nil)
0

_$ (vl-list-length '(1 2))
2

_$ (vl-list-length '(1 2 . 3))
nil
```

See Also

The listp function.

vl-list-loaded-vlx

Returns a list of all separate-namespace VLX files associated with the current document.

```
(vl-list-loaded-vlx)
```

Return Values

A list of symbols identifying separate-namespace VLX applications associated with the current AutoCAD document, or `nil` if there are no VLX applications associated with the current document.

The **`vl-list-loaded-vlx`** function does not identify VLX applications that are loaded in the current document's namespace.

Examples

Test for loaded VLX files associated with the current AutoCAD document:

```
_$ (vl-list-loaded-vlx)
nil
```

No VLX files are associated with the current document.

Load two VLX files; both VLX applications have been compiled to run in their own namespace:

```
_$ (load "c:/my documents/visual lisp/examples/foo1.vlx")
nil
```

```
_$ (load "c:/my documents/visual lisp/examples/foo2.vlx")
nil
```

Test for loaded VLX files associated with the current AutoCAD document:

```
_$ (vl-list-loaded-vlx)
(FOO1 FOO2)
```

The two VLX files just loaded are identified by **`vl-list-loaded-vlx`**.

Load a VLX that was compiled to run in a document's namespace:

```
_$ (load "c:/my documents/visual lisp/examples/foolocal.vlx")
nil
```

Test for loaded VLX files:

```
_$ (vl-list-loaded-vlx)
(FOO1 FOO2))
```

The last VLX loaded (*foolocal.vlx*) is not returned by **`vl-list-loaded-vlx`** because the application was loaded into the document's namespace; the VLX does not have its own namespace.

vl-load-all

Loads a file into all open AutoCAD documents, and into any document subsequently opened during the current AutoCAD session.

```
(vl-load-all filename)
```

Arguments

filename	A string naming the file to be loaded. If the file is in the AutoCAD Support File Search Path, you can omit the path name, but you must always specify the file extension; **`vl-load-all`** does not assume a file type.

Return Values

Unspecified. If *filename* is not found, **`vl-load-all`** issues an error message.

Examples

```
_$ (vl-load-all "c:/my documents/visual lisp/examples/whichns.lsp")
nil
```

```
_$ (vl-load-all "yinyang.lsp")
nil
```

vl-load-com

Loads Visual LISP extensions to AutoLISP

```
(vl-load-com)
```

This function loads the extended AutoLISP functions provided with Visual LISP. The Visual LISP extensions implement ActiveX and AutoCAD reactor support through AutoLISP, and also provide ActiveX utility and data conversion functions, dictionary handling functions, and curve measurement functions.

If the extensions are already loaded, **`vl-load-com`** does nothing.

Return Values

Unspecified.

See Also

The load function in this reference and the "Using Extended AutoLISP Functions" topic in the *Visual LISP Developer's Guide*.

vl-load-reactors

Loads reactor support functions

`(vl-load-reactors)`

This function is identical to **`vl-load-com`** and is maintained for backward compatibility.

See Also

The vl-load-com function.

vl-member-if

Determines if the predicate is true for one of the list members

`(vl-member-if` *`predicate-function list`*`)`

The **`vl-member-if`** function passes each element in *list* to the function specified in *predicate-function*. If *predicate-function* returns a non-`nil` value, **`vl-member-if`** returns the rest of the list in the same manner as the member function.

Arguments

predicate-function	The test function. This can be any function that accepts a single argument and returns `T` for any user-specified condition. The *predicate-function* value can take one of the following forms: ■ *A symbol (function name)* ■ `'(LAMBDA (A1 A2) ...)` ■ `(FUNCTION (LAMBDA (A1 A2) ...))`
list	A list to be tested.

Return Values

A list, starting with the first element that passes the test and containing all elements following this in the original argument. If none of the elements passes the test condition, **`vl-member-if`** returns `nil`.

Examples

The following command draws a line:

```
_$ (COMMAND "_.LINE" '(0 10) '(30 50) nil)
nil
```

The following command uses **`vl-member-if`** to return association lists describing an entity, if the entity is a line:

```
_$ (vl-member-if
'(lambda (x) (= (cdr x) "AcDbLine"))
   (entget (entlast)))
((100 . "AcDbLine") (10 0.0 10.0 0.0) (11 30.0 50.0 0.0) (210 0.0
0.0 1.0))
```

See Also

The vl-member-if-not function.

vl-member-if-not

Determines if the predicate is `nil` for one of the list members

`(vl-member-if-not` ***`predicate-function list`*****`)`**

The **`vl-member-if-not`** function passes each element in *list* to the function specified in *predicate-function*. If the function returns `nil`, **`vl-member-if-not`** returns the rest of the list in the same manner as the **`member`** function.

Arguments

predicate-function	The test function. This can be any function that accepts a single argument and returns `T` for any user-specified condition. The *predicate-function* value can take one of the following forms: ■ *A symbol (function name)* ■ `'(LAMBDA (A1 A2) ...)` ■ `(FUNCTION (LAMBDA (A1 A2) ...))`
list	A list to be tested.

Return Values

A list, starting with the first element that fails the test and containing all elements following this in the original argument. If none of the elements fails the test condition, **`vl-member-if-not`** returns `nil`.

Examples

```
_$ (vl-member-if-not 'atom '(1 "Str" (0 . "line") nil t))
((0 . "line") nil T)
```

See Also

The vl-member-if function.

vl-position

Returns the index of the specified list item

`(vl-position` ***`symbol list`*****`)`**

Arguments

symbol	Any AutoLISP symbol.
list	A true list.

Return Values

An integer containing the index position of *symbol* in *list*, or `nil` if *symbol* does not exist in the list.

Note that the first list element is index 0, the second element is index 1, and so on.

Examples

```
_$ (setq stuff (list "a" "b" "c" "d" "e"))
("a" "b" "c" "d" "e")

_$ (vl-position "c" stuff)
2
```

vl-prin1-to-string

Returns the string representation of LISP data as if it were output by the prin1 function

```
(vl-prin1-to-string data)
```

Arguments

data	Any AutoLISP data.

Return Values

A string containing the printed representation of *data* as if displayed by **prin1**.

Examples

```
_$ (vl-prin1-to-string "abc")
"\"abc\""
```

```
_$ (vl-prin1-to-string "c:\\acadwin")
"\"C:\\\\ACADWIN\""
```

```
_$ (vl-prin1-to-string 'my-var)
"MY-VAR"
```

See Also

The vl-princ-to-string function.

vl-princ-to-string

Returns the string representation of LISP data as if it were output by the princ function

```
(vl-princ-to-string data)
```

Arguments

data	Any AutoLISP data.

Return Values

A string containing the printed representation of *data* as if displayed by **princ**.

Examples

```
_$ (vl-princ-to-string "abc")
"abc"
```

```
_$ (vl-princ-to-string "c:\\acadwin")
"C:\\ACADWIN"
```

```
_$ (vl-princ-to-string 'my-var)
"MY-VAR"
```

See Also

The vl-prin1-to-string function.

vl-propagate

Copies the value of a variable into all open document namespaces (and sets its value in any subsequent drawings opened during the current AutoCAD session)

(vl-propagate '*symbol*)

Arguments

symbol	A symbol naming an AutoLISP variable.

Return Values

Unspecified.

Examples

Command: **(vl-propagate 'radius)**
nil

vl-registry-delete

Deletes the specified key or value from the Windows registry

(vl-registry-delete *reg-key [val-name]*)

Arguments

reg-key	A string specifying a Windows registry key.
val-name	A string containing the value of the *reg-key* entry.

If *val-name* is supplied and is not `nil`, the specified value will be purged from the registry. If *val-name* is absent or `nil`, the function deletes the specified key and all of its values.

Return Values

`T` if successful, otherwise `nil`.

Examples

```
_$ (vl-registry-write "HKEY_CURRENT_USER\\Test" "" "test data")
"test data"

_$ (vl-registry-read "HKEY_CURRENT_USER\\Test")
"test data"

_$ (vl-registry-delete "HKEY_CURRENT_USER\\Test")
T
```

NOTE This function cannot delete a key that has subkeys. To delete a subtree you must use **`vl-registry-descendents`** to enumerate all subkeys and delete all of them.

See Also

The vl-registry-descendents, vl-registry-read, and vl-registry-write functions.

vl-registry-descendents

Returns a list of subkeys or value names for the specified registry key

`(vl-registry-descendents `*`reg-key [val-names]`*`)`

Arguments

reg-key	A string specifying a Windows registry key.
val-names	A string containing the values for the *reg-key* entry.

If *val-names* is supplied and is not `nil`, the specified value names will be listed from the registry. If *val-name* is absent or `nil`, the function displays all subkeys of *reg-key*.

Return Values

A list of strings, if successful, otherwise `nil`.

Examples

```
_$ (vl-registry-descendents "HKEY_LOCAL_MACHINE\\SOFTWARE")
("Description"  "Program Groups" "ORACLE" "ODBC" "Netscape"
"Microsoft")
```

See Also

The vl-registry-delete, vl-registry-read and vl-registry-write functions.

vl-registry-read

Returns data stored in the Windows registry for the specified key/value pair

(vl-registry-read *reg-key [val-name]*)

Arguments

reg-key	A string specifying a Windows registry key.
val-name	A string containing the value of a registry entry.

If *val-name* is supplied and is not `nil`, the specified value will be read from the registry. If *val-name* is absent or `nil`, the function reads the specified key and all of its values.

Return Values

A string containing registry data, if successful, otherwise `nil`.

Examples

```
_$  (vl-registry-read "HKEY_CURRENT_USER\\Test")
nil

_$ (vl-registry-write "HKEY_CURRENT_USER\\Test" "" "test data")
"test data"

_$  (vl-registry-read "HKEY_CURRENT_USER\\Test")
"test data"
```

See Also

The vl-registry-delete, vl-registry-descendents, and vl-registry-write functions.

vl-registry-write

Creates a key in the Windows registry

(vl-registry-write *reg-key [val-name val-data]*)

Arguments

reg-key	A string specifying a Windows registry key. **NOTE** You cannot use **vl-registry-write** for HKEY_USERS or KEY_LOCAL_MACHINE.
val-name	A string containing the value of a registry entry.
val-data	A string containing registry data.

If *val-name* is not supplied or is `nil`, a default value for the key is written. If *val-name* is supplied and *val-data* is not specified, an empty string is stored.

Return Values

vl-registry-write returns *val-data*, if successful, `nil` otherwise.

Examples

```
_$ (vl-registry-write "HKEY_CURRENT_USER\\Test" "" "test data")
"test data"
```

```
_$  (vl-registry-read "HKEY_CURRENT_USER\\Test")
"test data"
```

See Also

The vl-registry-delete, vl-registry-descendents, and vl-registry-read functions.

vl-remove

Removes elements from a list

(vl-remove *element-to-remove list*)

Arguments

element-to-remove	The value of the element to be removed; may be any LISP data type.
list	Any list.

Return Values

The *list* with all elements except those equal to *element-to-remove*.

Examples

```
_$ (vl-remove pi (list pi t 0 "abc"))
(T 0 "abc")
```

vl-remove-if

Returns all elements of the supplied list which fail the test function

(vl-remove-if *predicate-function list*)

Arguments

predicate-function	The test function. This can be any function that accepts a single argument and returns `T` for any user-specified condition. The *predicate-function* value can take one of the following forms: ■ *A symbol (function name)* ■ `'(LAMBDA (A1 A2) ...)` ■ `(FUNCTION (LAMBDA (A1 A2) ...))`
list	A list to be tested.

Return Values

A list containing all elements of *list* for which *predicate-function* returns `nil`.

Examples

```
_$ (vl-remove-if 'vl-symbolp (list pi t 0 "abc"))
(3.14159 0 "abc")
```

vl-remove-if-not

Returns all elements of the supplied list which pass the test function

(vl-remove-if-not *predicate-function list*)

Arguments

predicate-function	The test function. This can be any function that accepts a single argument and returns `T` for any user-specified condition. The *predicate-function* value can take one of the following forms: ■ *A symbol (function name)* ■ `'(LAMBDA (A1 A2) ...)` ■ `(FUNCTION (LAMBDA (A1 A2) ...))`
list	A list to be tested.

Return Values

A list containing all elements of *list* for which *predicate-function* returns a non-`nil` value

Examples

```
_$ (vl-remove-if-not 'vl-symbolp (list pi t 0 "abc"))
(T)
```

vl-some

Checks whether the predicate is not `nil` for one element combination

`(vl-some` *`predicate-function list [list]...`*`)`

Arguments

predicate-function	The test function. This can be any function that accepts as many arguments as there are lists provided with **`vl-some`**, and returns `T` on a user-specified condition. The *predicate-function* value can take one of the following forms: ■ *A symbol (function name)* ■ `'(LAMBDA (A1 A2) ...)` ■ `(FUNCTION (LAMBDA (A1 A2) ...))`
list	A list to be tested.

The **vl-some** function passes the first element of each supplied list as an argument to the test function, then the next element from each list, and so on. Evaluation stops as soon as the predicate function returns a non-`nil` value for an argument combination, or until all elements have been processed in one of the lists.

Return Values

The predicate value, if *predicate-function* returned a value other than `nil`, otherwise `nil`.

Examples

The following example checks whether `nlst` (a number list) has equal elements in sequence:

```
_$ (setq nlst (list 0 2 pi pi 4))
(0 2 3.14159 3.14159 4)

_$ (vl-some '= nlst (cdr nlst))
T
```

vl-sort

Sorts the elements in a list according to a given compare function

(vl-sort *list comparison-function*)

Arguments

list	Any list.
comparison-function	A comparison function. This can be any function that accepts two arguments and returns `T` (or any non-`nil` value) if the first argument precedes the second in the sort order. The *comparison-function* value can take one of the following forms:

- *A symbol (function name)*
- `'(LAMBDA (A1 A2) ...)`
- `(FUNCTION (LAMBDA (A1 A2) ...))`

Return Values

A list containing the elements of *list* in the order specified by *comparison-function*. Duplicate elements may be eliminated from the list.

Examples

Sort a list of numbers:

```
_$ (vl-sort '(3 2 1 3) '<)
(1 2 3)     ;
```

Note that the result list contains only one 3.

Sort a list of 2D points by Y coordinate:

```
_$ (vl-sort '((1 3) (2 2) (3 1))
             (function (lambda (e1 e2)
                         (< (cadr e1) (cadr e2)) ) ) )
((3 1) (2 2) (1 3))
```

Sort a list of symbols:

```
_$ (vl-sort
   '(a d c b a)
   '(lambda (s1 s2)
    (< (vl-symbol-name s1) (vl-symbol-name s2)) ) )
(A B C D)      ;  Note that only one A remains in the result list
```

vl-sort-i

Sorts the elements in a list according to a given compare function, and returns the element index numbers

(vl-sort-i list *comparison-function*)

Arguments

list	Any list.
comparison-function	A comparison function. This can be any function that accepts two arguments and returns `T` (or any non-`nil` value) if the first argument precedes the second in the sort order. The *comparison-function* value can take one of the following forms: ■ *A symbol (function name)* ■ `'(LAMBDA (A1 A2) ...)` ■ `(FUNCTION (LAMBDA (A1 A2) ...))`

Return Values

A list containing the index values of the elements of *list*, sorted in the order specified by *comparison-function*. Duplicate elements will be retained in the result.

Examples

Sort a list of characters in descending order:

```
_$ (vl-sort-i '("a" "d" "f" "c") '>)
(2 1 3 0)
```

The sorted list order is "f" "d" "c" "a"; "f" is the 3rd element (index 2) in the original list, "d" is the 2nd element (index 1) in the list, and so on.

Sort a list of numbers in ascending order:

```
_$ (vl-sort-i '(3 2 1 3) '<)
(2 1 3 0)
```

Note that both occurrences of 3 are accounted for in the result list.

Sort a list of 2D points by *Y* coordinate:

```
_$ (vl-sort-i '((1 3) (2 2) (3 1))
         (function (lambda (e1 e2)
                (< (cadr e1) (cadr e2)) ) ) )
(2 1 0)
```

Sort a list of symbols:

```
_$ (vl-sort-i
   '(a d c b a)
   '(lambda (s1 s2)
    (< (vl-symbol-name s1) (vl-symbol-name s2)) ) )
(4 0 3 2 1)
```

Note that both a's are accounted for in the result list.

vl-string->list

Converts a string into a list of character codes

```
(vl-string->list string)
```

Arguments

string	A string.

Return Values

A list, each element of which is an integer representing the character code of the corresponding character in *string*.

Examples

```
_$ (vl-string->list "")
nil
```

```
_$ (vl-string->list "12")
(49 50)
```

See Also

The vl-list->string function.

vl-string-elt

Returns the ASCII representation of the character at a specified position in a string

```
(vl-string-elt string position)
```

Arguments

string	A string to be inspected.
position	A displacement in the string; the first character is displacement 0. Note that an error occurs if *position* is outside of the range of the string.

Return Values

An integer denoting the ASCII representation of the character at the specified position

Examples

```
_$ (vl-string-elt "May the Force be with you" 8)
70
```

vl-string-left-trim

Removes the specified characters from the beginning of a string

```
(vl-string-left-trim character-set string)
```

Arguments

character-set	A string listing the characters to be removed.
string	The string to be stripped of *character-set*.

Return Values

A string containing a substring of *string* with all leading characters in *character-set* removed

Examples

```
_$ (vl-string-left-trim " \t\n" "\n\t STR ")
"STR "

_$ (vl-string-left-trim "12456789" "12463CPO is not R2D2")
"3CPO is not R2D2"

_$ (vl-string-left-trim " " "     There are too many spaces here")
"There are too many spaces here"
```

vl-string-mismatch

Returns the length of the longest common prefix for two strings, starting at specified positions

(vl-string-mismatch *str1 str2 [pos1 pos2 ignore-case-p]*)

Arguments

str1	The first string to be matched.
str2	The second string to be matched.
pos1	An integer identifying the position to search from in the first string; 0 if omitted.
pos2	An integer identifying the position to search from in the second string; 0 if omitted.
ignore-case-p	If `T` is specified for this argument, case is ignored, otherwise case is considered.

Return Values

An integer.

Examples

```
_$ (vl-string-mismatch "VL-FUN" "VL-VAR")
3

_$ (vl-string-mismatch "vl-fun" "avl-var")
0
```

```
_$ (vl-string-mismatch "vl-fun" "avl-var" 0 1)
3

_$ (vl-string-mismatch "VL-FUN" "Vl-vAR")
1

_$ (vl-string-mismatch "VL-FUN" "Vl-vAR" 0 0 T)
3
```

vl-string-position

Looks for a character with the specified ASCII code in a string

(vl-string-position *char-code str [start-pos [from-end-p]]*)

Arguments

char-code	The integer representation of the character to be searched.
str	The string to be searched.
start-pos	The position to begin searching from in the string (first character is 0); 0 if omitted.
from-end-p	If `T` is specified for this argument, the search begins at the end of the string and continues backward to *pos*.

Return Values

An integer representing the displacement at which *char-code* was found from the beginning of the string; `nil` if the character was not found.

Examples

```
_$ (vl-string-position (ascii "z") "azbdc")
1

_$ (vl-string-position 122 "azbzc")
1

_$ (vl-string-position (ascii "x") "azbzc")
nil
```

The search string used in the following example contains two “z” characters. Reading from left to right, with the first character being displacement 0, there is one z at displacement 1 and another z at displacement 3:

```
_$ (vl-string-position (ascii "z") "azbzlmnqc")
1
```

Searching from left to right (the default), the "z" in position 1 is the first one **vl-string-position** encounters. But when searching from right to left, as in the following example, the "z" in position 3 is the first one encountered:

```
_$ (vl-string-position (ascii "z") "azbzlmnqc" nil t)
3
```

vl-string-right-trim

Removes the specified characters from the end of a string

(vl-string-right-trim *character-set string*)

Arguments

character-set	A string listing the characters to be removed.
string	The string to be stripped of *character-set*.

Return Values

A string containing a substring of *string* with all trailing characters in *character-set* removed.

Examples

```
_$ (vl-string-right-trim " \t\n" " STR \n\t ")
" STR"

_$ (vl-string-right-trim "1356789" "3CPO is not R2D267891")
"3CPO is not R2D2"

_$ (vl-string-right-trim " " "There are too many spaces here     ")
"There are too many spaces here"
```

vl-string-search

Searches for the specified pattern in a string

(vl-string-search *pattern string [start-pos]*)

Arguments

pattern	A string containing the pattern to be searched for.
string	The string to be searched for *pattern*.

start-pos	An integer identifying the starting position of the search; 0, if omitted.

Return Values

An integer representing the position in the string where the specified *pattern* was found, or `nil` if the pattern is not found; the first character of the string is position 0.

Examples

```
_$ (vl-string-search "foo" "pfooyey on you")
1

_$ (vl-string-search "who" "pfooyey on you")
nil

_$ (vl-string-search "foo" "fooey-more-fooey" 1)
11
```

vl-string-subst

Substitutes one string for another, within a string

`(vl-string-subst` ***`new-str pattern string [start-pos]`*** **`)`**

Arguments

new-str	The string to be substituted for *pattern*.
pattern	A string containing the pattern to be replaced.
string	The string to be searched for *pattern*.
start-pos	An integer identifying the starting position of the search; 0, if omitted.

Note that the search is case-sensitive, and **`vl-string-subst`** only substitutes the first occurrence it finds of the string.

Return Values

The value of *string* after any substitutions have been made

Examples

Replace the string "Ben" with "Obi-wan":

```
_$ (vl-string-subst "Obi-wan" "Ben" "Ben Kenobi")
"Obi-wan Kenobi"
```

Replace “Ben” with “Obi-wan”:

```
_$ (vl-string-subst "Obi-wan" "Ben" "ben Kenobi")
"ben Kenobi"
```

Nothing was substituted because **vl-string-subst** did not find a match for “Ben”; the “ben” in the string that was searched begins with a lowercase “b”.

Replace “Ben” with “Obi-wan”:

```
_$ (vl-string-subst "Obi-wan" "Ben" "Ben Kenobi Ben")
"Obi-wan Kenobi Ben"
```

Note that there are two occurrences of “Ben” in the string that was searched, but **vl-string-subst** only replaces the first occurrence.

Replace “Ben” with “Obi-wan,” but start the search at the fourth character in the string:

```
_$ (vl-string-subst "Obi-wan" "Ben" "Ben \"Ben\" Kenobi" 3)
"Ben \"Obi-wan\" Kenobi"
```

There are two occurrences of “Ben” in the string that was searched, but because **vl-string-subst** was instructed to begin searching at the fourth character, it found and replaced the second occurrence, not the first.

vl-string-translate

Replaces characters in a string with a specified set of characters

(vl-string-translate *source-set dest-set str*)

Arguments

source-set	A string of characters to be matched.
dest-set	A string of characters to be substituted for those in *source-set*.
str	A string to be searched and translated.

Return Values

The value of *str* after any substitutions have been made

Examples

```
_$ (vl-string-translate "abcABC" "123123" "A is a, B is b, C is C")
"1 is 1, 2 is 2, 3 is 3"

_$ (vl-string-translate "abc" "123" "A is a, B is b, C is C")
"A is 1, B is 2, C is C"
```

vl-string-trim

Removes the specified characters from the beginning and end of a string

```
(vl-string-trim char-set str)
```

Arguments

char-set	A string listing the characters to be removed.
str	The string to be trimmed of *char-set*.

Return Values

The value of *str*, after any characters have been trimmed.

Examples

```
_$ (vl-string-trim " \t\n" " \t\n STR \n\t ")
"STR"

_$ (vl-string-trim "this is junk" "this is junk Don't call this
junk! this is junk")
"Don't call this junk!"

_$ (vl-string-trim " " "       Leave me alone        ")
"Leave me alone"
```

vl-symbol-name

Returns a string containing the name of a symbol

```
(vl-symbol-name  symbol)
```

Arguments

symbol	Any LISP symbol.

Return Values

A string containing the name of the supplied symbol argument, in uppercase.

Examples

```
_$ (vl-symbol-name 'S::STARTUP)
"S::STARTUP"

_$ (progn (setq sym 'my-var) (vl-symbol-name sym))
"MY-VAR"

_$ (vl-symbol-name 1)
; *** ERROR: bad argument type: symbolp 1
```

vl-symbol-value

Returns the current value bound to a symbol

```
(vl-symbol-value symbol)
```

This function is equivalent to the **eval** function, but does not call the LISP evaluator.

Arguments

symbol	Any LISP symbol.

Return Values

The value of *symbol*, after evaluation.

Examples

```
_$ (vl-symbol-value 't)
T

_$ (vl-symbol-value 'PI)
3.14159

_$ (progn (setq sym 'PAUSE) (vl-symbol-value sym))
"\\"
```

vl-symbolp

Identifies whether or not a specified object is a symbol

Arguments

(vl-symbolp *object*)

object	Any LISP object.

Return Values

`T` if *object* is a symbol, otherwise `nil`.

Examples

```
_$ (vl-symbolp t)
T

_$ (vl-symbolp nil)
nil

_$ (vl-symbolp 1)
nil

_$ (vl-symbolp (list 1))
nil
```

vl-unload-vlx

Unload a VLX application that is loaded in its own namespace

(vl-unload-vlx *appname*)

Arguments

appname	A string naming a VLX application that is loaded in its own namespace. Do not include the *.vlx* extension.

The **`vl-unload-vlx`** function does not unload VLX applications that are loaded in the current document's namespace.

Return Values

`T` if successful, otherwise **`vl-unload-vlx`** results in an error.

Examples

Assuming that *vlxns* is an application that is loaded in its own namespace, the following command unloads *vlxns*:

```
Command: (vl-unload-vlx "vlxns")
T
```

Try unloading **`vlxns`** again:

```
Command: (vl-unload-vlx "vlxns")
; *** ERROR: LISP Application is not found VLXNS
```

The **`vl-unload-vlx`** command fails this time, because the application was not loaded.

See Also

The load and vl-vlx-loaded-p functions.

vl-vbaload

Loads a Visual Basic project

Arguments

(vl-vbaload *filename*)

filename	A string naming the Visual Basic project file to be loaded.

Return Values

Unspecified, if successful.

Examples

```
_$ (vl-vbaload "c:/program files/autocad 2000i/sample/vba/
drawline.dvb")
"c:\\program files\\autocad 2000i\\sample\\vba\\drawline.dvb"
```

See Also

The vl-vbarun function.

vl-vbarun

Runs a Visual Basic macro

Arguments

(vl-vbarun *macroname*)

macroname	A string naming a loaded Visual Basic macro.

Return Values

macroname

Examples

Load a VBA project file:

```
_$ (vl-vbaload "c:/program files/autocad 2000i/sample/vba/
drawline.dvb")
"c:\\program files\\autocad 2000i\\sample\\vba\\drawline.dvb"
```

Run a macro from the loaded project:

```
_$ (vl-vbarun "drawline")
"drawline"
```

See Also

The vl-vbaload function.

vl-vlx-loaded-p

Determines whether a separate-namespace VLX is currently loaded

(vl-vlx-loaded-p *appname*)

Arguments

appname	A string naming a VLX application.

Return Values

`T` if the application is loaded, `nil` if it is not loaded.

Examples

Check to see if the **`vlxns`** application is loaded in its own namespace:

```
Command: (vl-vlx-loaded-p "vlxns")
nil
```

The application is not loaded in its own namespace.

Now load **`vlxns`**:

```
Command: (load "vlxns.vlx")
nil
```

Check to see if the **`vlxns`** application loaded successfully:

```
Command: (vl-vlx-loaded-p "vlxns")
T
```

This example assumes **`vlxns`** was defined to run in its own namespace. If the application was not defined to run in its own namespace, it would load into the current document's namespace and **`vl-vlx-loaded-p`** would return `nil`.

See Also

The load and vl-unload-vlx functions.

vlax-3D-point

Creates ActiveX-compatible (variant) 3D point structure

`(vlax-3D-point` ***`list`*** **`)`** or **`(vlax-3D-point`** ***`x y`*** **`[`*`z`*`])`**

Arguments

list	A list of 2 or 3 numbers, representing points.
x, *y*	Numbers representing *X* and *Y* coordinates of a point.
z	A number representing the *Z* coordinate of a point.

Return Values

A variant containing a three-element array of doubles.

Examples

```
_$ (vlax-3D-point 5 20)
#<variant 8197 ...>
```

```
_$ (vlax-3D-point '(33.6 44.0 90.0))
<variant 8197 ...>
```

See Also

The vlax-make-safearray, vlax-make-variant, vlax-safearray-fill, and vlax-safearray-put-element functions.

vlax-add-cmd

Adds commands to the AutoCAD built-in command set

(vlax-add-cmd *global-name func-sym [local-name cmd-flags]*)

With **vlax-add-cmd** you can define a function as an AutoCAD command, without using the *c:* prefix in the function name. You can also define a transparent AutoLISP command, which is not possible with a *c:* function.

WARNING! You cannot use the **command** function call in a transparently-defined **vlax-add-cmd** function. Doing so can cause AutoCAD to close unexpectedly.

The **vlax-add-cmd** function makes an AutoLISP function visible as an ObjectARX-style command at the AutoCAD Command prompt during the current AutoCAD session. The function provides access to the ObjectARX acedRegCmds macro, which provides a pointer to the ObjectARX system AcEdCommandStack object.

The **vlax-add-cmd** function automatically assigns commands to command groups. When issued from a document namespace, **vlax-add-cmd** adds the command to a group named *doc-ID*; *doc-ID* is a hexadecimal value identifying the document. If issued from a separate-namespace VLX, **vlax-add-cmd** adds the command to a group named VLC-D*doc-ID*:*VLX-name*, where *VLX-name* is the name of the application that issued **vlax-add-cmd**.

It is recommended that you use the **`vlax-add-cmd`** function from a separate-namespace VLX. You should then explicitly load the VLX using the APPLOAD command, rather than by placing it in one of the startup LISP files.

NOTE You cannot use **`vlax-add-cmd`** to expose functions that create reactor objects or serve as reactor callbacks.

Arguments

global-name	A string.
func-sym	A symbol naming an AutoLISP function with zero arguments.
local-name	A string (defaults to *global-name*).
cmd-flags	An integer (defaults to ACRX_CMD_MODAL + ACRX_CMD_REDRAW)

The primary flags are:

ACRX_CMD_MODAL (0) Command cannot be invoked while another command is active.

ACRX_CMD_TRANSPARENT (1) Command can be invoked while another command is active.

The secondary flags are:

ACRX_CMD_USEPICKSET (2) When the pickfirst set is retrieved it is cleared within AutoCAD. Command will be able to retrieve the pickfirst set. Command cannot retrieve or set grips.

ACRX_CMD_REDRAW (4) When the pickfirst set or grip set is retrieved, neither will be cleared within AutoCAD. Command can retrieve the pickfirst set and the grip set.

If both ACRX_CMD_USEPICKSET and ACRX_CMD_REDRAW are set, the effect is the same as if just ACRX_CMD_REDRAW is set. For more information on the flags, refer to the Command Stack topic in the *ObjectARX Reference* manual.

Return Values

The *global-name* argument, if successful. The function returns `nil`, if acedRegCmds->addCommand(...) returns an error condition.

Examples

The **hello-autocad** function in the following example has no c: prefix, but **vlax-add-cmd** makes it visible as an ObjectARX-style command at the AutoCAD Command prompt:

```
_$ (defun hello-autocad () (princ "hello Visual LISP"))
HELLO-AUTOCAD

_$ (vlax-add-cmd "hello-autocad" 'hello-autocad)
"hello-autocad"
```

See Also

The vlax-remove-cmd function.

vlax-create-object

Creates a new instance of an application object

(vlax-create-object *prog-id*)

Use **vlax-create-object** when you want a new instance of an application to be started, and an object of the type specified by <*Component*> (see the argument description) to be created. To use the current instance, use **vlax-get-object**. However, if an application object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times you call **vlax-create-object**.

Arguments

prog-id	A string containing the programmatic identifier of the desired ActiveX object. The format of *prog-id* is: <*Vendor*>.<*Component*>.<*Version*> For example: `AutoCAD.Drawing.15`

Return Values

The application object (VLA-object).

Examples

Create an instance of an Excel application:

```
_$  (vlax-create-object "Excel.Application")
#<VLA-OBJECT _Application 0017b894>
```

vlax-curve-getArea

Returns the area inside the curve

```
(vlax-curve-getArea curve-obj)
```

Arguments

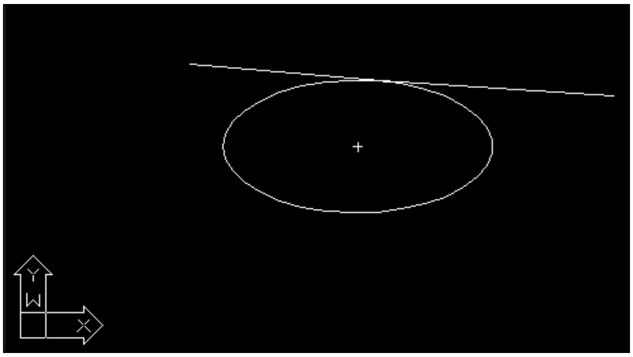
curve-obj	The VLA-object to be measured.

Return Values

A real number representing the area of the curve, if successful, otherwise `nil`.

Examples

Assume the curve being measured is the ellipse in the following drawing:



Sample curve (ellipse) for vlax-curve-getarea

The `ellipseObj` variable points to the ellipse VLA-object.

The following command obtains the area of the curve:

```
_$ (vlax-curve-getArea ellipseObj)
4.712393
```

vlax-curve-getClosestPointTo

Returns the point (in WCS) on a curve that is nearest to the specified point

(vlax-curve-getClosestPointTo ***curve-obj givenPnt [extend]*****)**

Arguments

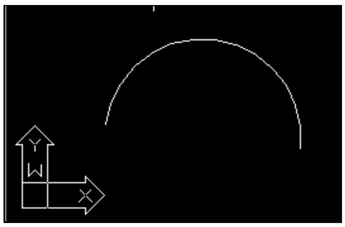
curve-obj	The VLA-object to be measured.
givenPnt	A point (in WCS) for which to find the nearest point on the curve.
extend	If specified and not `nil`, **`vlax-curve-getClosestPointTo`** extends the curve when searching for the nearest point.

Return Values

A 3D point list representing a point on the curve, if successful, otherwise `nil`.

Examples

Assume that the curve being measured is the arc in the following drawing:



Return the closest point on the arc to the coordinates 6.0, 0.5:

```
_$ (vlax-curve-getClosestPointTo arcObj '(6.0 0.5 0.0))
(6.0 1.5 0.0)
```

Return the closest point on the arc to the coordinates 6.0, 0.5, after extending the arc:

```
_$ (vlax-curve-getClosestPointTo arcObj '(6.0 0.5 0.0) T)
(5.7092 0.681753 0.0)
```

vlax-curve-getClosestPointToProjection

Returns the closest point (in WCS) on a curve after projecting the curve onto a plane

```
(vlax-curve-getClosestPointToProjection curve-obj
  givenPnt normal [extend])
```

Arguments

curve-obj	The VLA-object to be measured.
givenPnt	A point (in WCS) for which to find the nearest point on the curve.
normal	*A* normal vector (in WCS) for the plane to project onto.
extend	If specified and not `nil`, **`vlax-curve-getClosestPointToProjection`** extends the curve when searching for the nearest point.

`vlax-curve-getClosestPointToProjection` projects the curve onto the plane defined by the *givenPnt* and *normal*, then calculates the nearest point on that projected curve to *givenPnt*. The resulting point is then projected back onto the original curve, and **`vlax-curve-getClosestPointToProjection`** returns that projected point.

Return Values

A 3D point list representing a point on the curve, if successful, otherwise `nil`.

vlax-curve-getDistAtParam

Returns the length of the curve's segment from the curve's beginning to the specified parameter

```
(vlax-curve-getDistAtParam curve-obj param)
```

Arguments

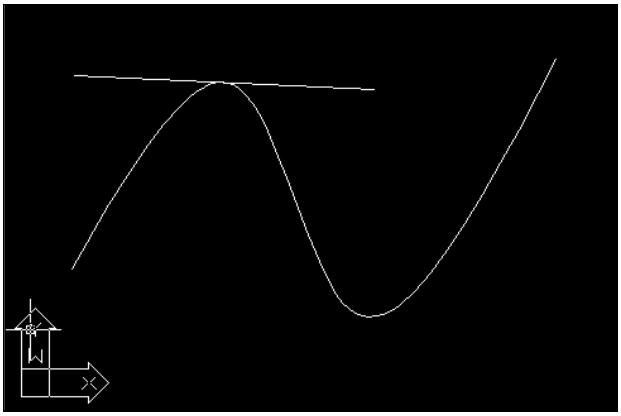
curve-obj	The VLA-object to be measured.
param	A number specifying a parameter on the curve.

Return Values

A real number that is the length up to the specified parameter, if successful, otherwise `nil`.

Examples

Assume that `splineObj` points to the spline in the following drawing:



Sample curve (spline) for vlax-curve-getDistAtParam

Obtain the start parameter of the curve:

```
_$ (setq startSpline (vlax-curve-getStartParam splineObj))
0.0
```

The curve starts at parameter 0.

Obtain the end parameter of the curve:

```
_$ (setq endSpline (vlax-curve-getEndParam splineObj))
17.1546
```

The curve's end parameter is 17.1546.

Determine the distance to parameter midway along the curve:

```
_$ (vlax-curve-getDistAtParam splineObj
   ( / (- endspline startspline) 2))
8.99417
```

The distance from the start to the middle of the curve is 8.99417.

vlax-curve-getDistAtPoint

Returns the length of the curve's segment between the curve's start point and the specified point

```
(vlax-curve-getDistAtPoint curve-obj point)
```

Arguments

curve-obj	The VLA-object to be measured.
point	A 3D point list (in WCS) on *curve-obj*.

Return Values

A real number if successful, otherwise `nil`.

Examples

For the following example, assume that `splineObj` points to the spline shown in the example for vlax-curve-getDistAtParam.

Set OSNAP to tangent and select the point where the line is tangent to the curve:

```
_$ (setq selPt (getpoint))
(4.91438 6.04738 0.0)
```

Determine the distance from the start of the curve to the selected point:

```
_$ (vlax-curve-getDistAtPoint splineObj selpt)
5.17769
```

vlax-curve-getEndParam

Returns the parameter of the endpoint of the curve

```
(vlax-curve-getEndParam curve-obj)
```

Arguments

curve-obj	The VLA-object to be measured.

Return Values

A real number representing an end parameter, if successful, otherwise `nil`.

Examples

Assuming that `ellipseObj` points to the ellipse shown in the example for vlax-curve-getArea, the following function call returns the end parameter of the curve:

```
_$ (vlax-curve-getendparam ellipseObj)
6.28319
```

The end parameter is 6.28319 (twice pi).

See Also

The vlax-curve-getStartParam function.

vlax-curve-getEndPoint

Returns the endpoint (in WCS) of the curve

`(vlax-curve-getEndPoint `*`curve-obj`*`)`

Arguments

curve-obj	The VLA-object to be measured.

Return Values

A 3D point list representing an endpoint, if successful, otherwise `nil`.

Examples

Get the endpoint of the ellipse used to demonstrate vlax-curve-getArea:

```
_$ (vlax-curve-getEndPoint ellipseObj)
(2.0 2.0 0.0)
```

vlax-curve-getFirstDeriv

Returns the first derivative (in WCS) of a curve at the specified location

`(vlax-curve-getFirstDeriv `*`curve-obj param`*`)`

Arguments

curve-obj	The VLA-object to be measured.
param	A number specifying a parameter on the curve.

Return Values

A 3D vector list, if successful, otherwise `nil`.

Examples

For the following example, assume that `splineObj` points to the spline shown in the example of the vlax-curve-getDistAtParam function.

Obtain the start parameter of the curve:

```
_$ (setq startSpline (vlax-curve-getStartParam splineObj))
0.0
```

Obtain the end parameter of the curve:

```
_$ (setq endSpline (vlax-curve-getEndParam splineObj))
17.1546
```

Determine the first derivative at the parameter midway along the curve:

```
_$ (vlax-curve-getFirstDeriv splineObj
   ( / (- endspline startspline) 2))
(0.422631 -1.0951 0.0)
```

vlax-curve-getParamAtDist

Returns the parameter of a curve at the specified distance from the beginning of the curve

(vlax-curve-getParamAtDist *curve-obj dist*)

Arguments

curve-obj	The VLA-object to be measured.
dist	A number specifying the distance from the beginning of the curve.

Return Values

A real number representing a parameter, if successful, otherwise `nil`.

Examples

Assuming that `splineObj` points to the spline shown in the example for vlax-curve-getDistAtParam, determine the parameter at a distance of 1.0 from the beginning of the spline:

```
_$ (vlax-curve-getParamAtDist splineObj 1.0)
0.685049
```

vlax-curve-getParamAtPoint

Returns the parameter of the curve at the point

(vlax-curve-getParamAtPoint *curve-obj point*)

Arguments

curve-obj	The VLA-object to be measured.
point	A 3D point list (in WCS) on *curve-obj*.

Return Values

A real number representing a parameter, if successful, otherwise `nil`.

Examples

Assuming that `ellipseObj` points to the ellipse shown in the example for vlax-curve-getArea, set OSNAP to tangent and select the point where the line is tangent to the ellipse:

```
_$ (setq selPt (getpoint))
(7.55765 5.55066 0.0)
```

Get the parameter value at the selected point:

```
_$ (vlax-curve-getParamAtPoint ellipseObj selPt)
4.58296
```

vlax-curve-getPointAtDist

Returns the point (in WCS) along a curve at the distance specified by the user

(vlax-curve-getPointAtDist *curve-obj dist*)

Arguments

curve-obj	The VLA-object to be measured.
dist	The distance along the curve from the beginning of the curve to the location of the specified point.

Return Values

A 3D point list representing a point on the curve, if successful, otherwise `nil`.

Examples

Assuming that `splineObj` points to the spline shown in the example for vlax-curve-getDistAtParam, determine the point at a distance of 1.0 from the beginning of the spline:

```
_$ (vlax-curve-getPointAtDist splineObj 1.0)
(2.24236 2.99005 0.0)
```

vlax-curve-getPointAtParam

Returns the point at the specified parameter value along a curve

(vlax-curve-getPointAtParam *curve-obj param*)

Arguments

curve-obj	The VLA-object to be measured.
param	A number specifying a parameter on the curve.

Return Values

A 3D point list representing a point on the curve, if successful, otherwise `nil`.

Examples

For the following example, assume that `splineObj` points to the spline shown in the example for vlax-curve-getDistAtParam.

Obtain the start parameter of the curve:

```
_$ (setq startSpline (vlax-curve-getStartParam splineObj))
0.0
```

Obtain the end parameter of the curve:

```
_$ (setq endSpline (vlax-curve-getEndParam splineObj))
17.1546
```

Determine the point at the parameter midway along the curve:

```
_$ (vlax-curve-getPointAtParam splineObj
   ( / (- endspline startspline) 2))
(6.71386 2.82748 0.0)
```

vlax-curve-getSecondDeriv

Returns the second derivative (in WCS) of a curve at the specified location

(vlax-curve-getSecondDeriv *curve-obj param*)

Arguments

curve-obj	The VLA-object to be measured.
param	A number specifying a parameter on the curve.

Return Values

A 3D vector list, if successful, otherwise `nil`.

Examples

For the following example, assume that `splineObj` points to the spline shown in the example of the vlax-curve-getDistAtParam function.

Obtain the start parameter of the curve:

```
_$ (setq startSpline (vlax-curve-getStartParam splineObj))
0.0
```

Obtain the end parameter of the curve:

```
_$ (setq endSpline (vlax-curve-getEndParam splineObj))
17.1546
```

Determine the second derivative at the parameter midway along the curve:

```
_$ (vlax-curve-getSecondDeriv splineObj
   ( / (- endspline startspline) 2))
(0.0165967 0.150848 0.0)
```

vlax-curve-getStartParam

Returns the start parameter on the curve

(vlax-curve-getStartParam *curve-obj*)

Arguments

curve-obj	The VLA-object to be measured.

Return Values

A real number representing the start parameter, if successful, otherwise `nil`.

Examples

Assuming that `ellipseObj` points to the ellipse shown in the example for vlax-curve-getArea, determine the start parameter of the curve:

```
_$ (vlax-curve-getstartparam ellipseObj)
0.0
```

See Also

The vlax-curve-getEndParam function.

vlax-curve-getStartPoint

Returns the start point (in WCS) of the curve

`(vlax-curve-getStartPoint` *`curve-obj`*`)`

Arguments

curve-obj	The VLA-object to be measured.

Return Values

A 3D point list representing the start point, if successful, otherwise `nil`.

Examples

Get the start point of the ellipse used to demonstrate vlax-curve-getArea:

```
_$ (vlax-curve-getStartPoint ellipseObj)
(2.0 2.0 0.0)
```

For an ellipse, the start points and endpoints are the same.

Obtain the start point of the spline used to demonstrate vlax-curve-getDistAtParam:

```
_$ (vlax-curve-getStartPoint splineObj)
(1.73962 2.12561 0.0)
```

vlax-curve-isClosed

Determines if the specified curve is closed (that is, the start point is the same as the end-point)

```
(vlax-curve-isClosed curve-obj)
```

Arguments

curve-obj	The VLA-object to be tested.

Return Values

`T` if the curve is closed, otherwise `nil`.

Examples

Determine if the ellipse used to demonstrate vlax-curve-getArea is closed:

```
_$ (vlax-curve-isClosed ellipseObj)
T
```

Determine if the spline used to demonstrate vlax-curve-getDistAtParam is closed:

```
_$ (vlax-curve-isClosed splineObj)
nil
```

vlax-curve-isPeriodic

Determines if the specified curve has an infinite range in both directions and there is a period value *dT,* such that a point on the curve at (*u* + *dT*) = point on curve (*u*), for any parameter *u*.

```
(vlax-curve-isPeriodic curve-obj)
```

Arguments

curve-obj	The VLA-object to be tested.

Return Values

`T` if the curve is periodic, otherwise `nil`.

Examples

Determine if the ellipse used to demonstrate vlax-curve-getArea is periodic:

```
_$ (vlax-curve-isPeriodic ellipseObj)
T
```

Determine if the spline used to demonstrate vlax-curve-getDistAtParam is periodic:

```
_$ (vlax-curve-isPeriodic splineObj)
nil
```

vlax-curve-isPlanar

Determines if there is a plane that contains the curve

(vlax-curve-isPlanar *curve-obj*)

Arguments

curve-obj	The VLA-object to be tested.

Return Values

`T` if there is a plane that contains the curve, otherwise `nil`.

Examples

Determine if there is a plane containing the ellipse used to demonstrate vlax-curve-getArea:

```
_$ (vlax-curve-isPlanar ellipseObj)
T
```

Determine if there is a plane containing the spline used to demonstrate vlax-curve-getDistAtParam:

```
_$ (vlax-curve-isPeriodic splineObj)
nil
```

vlax-dump-object

Lists an object's properties, and optionally, the methods that apply to the object

(vlax-dump-object *obj* [T])

Arguments

obj	A VLA-object.
T	If specified, **`vlax-dump-object`** also lists all methods that apply to *obj*.

Return Values

`T`, if successful. If an invalid object name is supplied, **`vlax-dump-object`** displays an error message.

Examples

```
_$ (setq aa (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00b3b91c>

_$ (vlax-dump-object aa)
; IAcadApplication: AutoCAD Application Interface
; Property values:
;   ActiveDocument (RO) = #<VLA-OBJECT IAcadDocument 01b52fac>
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00b3b91c>
;   Caption (RO) = "AutoCAD - [Drawing.dwg]"
.
.
.
T
```

List an object's properties and the methods that apply to the object:

```
_$ (vlax-dump-object aa T)
; IAcadApplication: AutoCAD Application Interface
; Property values:
;   ActiveDocument (RO) = #<VLA-OBJECT IAcadDocument 01b52fac>
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00b3b91c>
;   Caption (RO) = "AutoCAD - [Drawing.dwg]"
.
.
.

; Methods supported:
;   EndUndoMark ()
;   Eval (1)
;   GetInterfaceObject (1)
;   ListAds ()
;   ListArx ()
.
.
.
T
```

vlax-ename->vla-object

Transforms entity to VLA-object

(vlax-ename->vla-object *entname*)

Arguments

entname	An entity name (ename data type).

Return Values

A VLA-object.

Examples

```
_$ (setq e (car (entsel)))
<Entity name: 27e0540>

_$ (vlax-ename->vla-object e)
#<VLA-OBJECT IAcadLWPolyline 03f713a0>
```

See Also

The vlax-vla-object->ename function.

vlax-erased-p

Determines whether an object was erased

(vlax-erased-p *obj)*

Arguments

obj	A VLA-object.

Return Values

`T` if the object was erased, otherwise `nil`.

vlax-for

Iterates through a collection of objects, evaluating each expression

(vlax-for *symbol collection [expression1 [expression2 ...]]*)

Arguments

symbol	A symbol to be assigned to each VLA-object in a collection.
collection	A VLA-object representing a collection object.
expression1, expression2...	The expressions to be evaluated.

Return Values

The value of the last expression evaluated for the last object in the collection.

Examples

The following code issues **vlax-dump-object** on every drawing object in the model space:

```
(vl-load-com)                        ; load ActiveX support
(vlax-for for-item
   (vla-get-modelspace
        (vla-get-activedocument (vlax-get-acad-object))
   )
  (vlax-dump-object for-item)        ; list object properties
)
```

vlax-get-acad-object

Retrieves the top level AutoCAD application object for the current AutoCAD session

(vlax-get-acad-object)

Return Values

A VLA-object.

Examples

```
_$ (setq aa (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00b3b91c>
```

vlax-get-object

Returns a running instance of an application object

```
(vlax-get-object prog-id)
```

Arguments

prog-id	A string that identifies the desired application object. The format of *prog-id* is: *appname.objecttype* where *appname* is the name of the application and *objecttype* is the application object. The *objecttype* may be followed by a version number. **NOTE** You can usually find the *prog-id* for an application in that application's online help. For example, Microsoft® Office applications document this information in the Visual Basic® Reference section of their online help.

Return Values

The application object, or `nil`, if there is no instance of the specified object currently running.

Examples

Obtain the Application object for the Excel program:

```
_$ (vlax-get-object "Excel.Application")
#<VLA-OBJECT _Application 0017bb5c>
```

vlax-get-or-create-object

Returns a running instance of an application object, or creates a new instance, if the application is not currently running

(vlax-get-or-create-object *prog-id*)

Arguments

prog-id	A string containing the programmatic identifier of the desired ActiveX object desired. The format of *prog-id* is: *<Vendor>.<Component>.<Version>* For example: `AutoCAD.Drawing.15`

Return Values

The object.

Examples

```
_$ (vlax-get-or-create-object "Excel.Application")
#<VLA-OBJECT _Application 0017bb5c>
```

vlax-get-property

Retrieves a VLA-object's property

(vlax-get-property *object property*)

This function was formerly known as vlax-get.

Arguments

object	A VLA-object.
property	A symbol or string naming the property to be retrieved.

Return Values

The value of the object's property.

Examples

Begin by retrieving a pointer to the root AutoCAD object:

```
_$ (setq acadObject (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00a4b2b4>
```

Get the AutoCAD ActiveDocument property:

```
_$ (setq acadDocument (vlax-get-property acadObject
'ActiveDocument))
#<VLA-OBJECT IAcadDocument 00302a18>
```

The function returns the current document object.

Get the ModelSpace property of the ActiveDocument object:

```
_$ (setq mSpace (vlax-get-property acadDocument 'Modelspace))
#<VLA-OBJECT IAcadModelSpace 00c14b44>
```

The model space object of the current document is returned.

Convert a drawing entity to a VLA-object:

```
_$ (setq vlaobj (vlax-ename->vla-object e))
#<VLA-OBJECT IAcadLWPolyline 0467114c>
```

Get the color property of the object:

```
_$ (vlax-get-property vlaobj 'Color)
256
```

See Also

The vlax-property-available-p and vlax-put-property functions.

vlax-import-type-library

Imports information from a type library

```
(vlax-import-type-library :tlb-filename filename
  [:methods-prefix mprefix  :properties-prefix pprefix
  :constants-prefix cprefix])
```

Arguments

filename	A string naming the type library. A file can be one of the following types: ■ *A type library (.tlb) or object library (.olb) file* ■ *An executable (.exe) file* ■ *A library (.dll) file containing a type library resource* ■ *A compound document holding a type library* ■ Any other file format that can be understood by the LoadTypeLib API If you omit the path from *tlb-filename*, AutoCAD looks for the file in the Support File Search Path.
mprefix	Prefix to be used for method wrapper functions. For example, if the type library contains a Calculate method and the *mprefix* parameter is set to "cc-", Visual LISP generates a wrapper function named **`cc-Calculate`**. This parameter defaults to "".
pprefix	Prefix to be used for property wrapper functions. For example, if the type library contains a Width property with both read and write permissions, and *pprefix* is set to "cc-", then Visual LISP generates wrapper functions named **`cc-get-Width`** and **`cc-put-Width`**. This parameter defaults to "".
cprefix	Prefix to be used for constants contained in the type library. For example, if the type library contains a ccMaxCountOfRecords property with both read and write permissions, and *cprefix* is set to "cc-", Visual LISP generates a constant named `cc-ccMaxCountOfRecords`. This parameter defaults to "".

Note the required use of keywords when passing arguments to **`vlax-import-type-library`**.

Return Values

`T`, if successful.

Examples

Import a Microsoft Word™ type library, assigning the prefix “msw-” to methods and properties, and “mswc-” to constants:

```
_$ (vlax-import-type-library
   :tlb-filename "c:/program files/microsoft office/msword8.olb"
   :methods-prefix "msw-"
   :properties-prefix "msw-"
   :constants-prefix "mswc-")
T
```

Remarks

Function wrappers created by **`vlax-import-type-library`** are available only in the context of the document **`vlax-import-type-library`** was issued from.

In the current release of Visual LISP, **`vlax-import-type-library`** is executed at runtime, rather than at compile-time. In future releases of Visual LISP, this may change. The following practices are recommended when using **`vlax-import-type-library`**:

- If you want your code to run on different machines, avoid specifying an absolute path in the *tlb-file-name* parameter.
- If possible, avoid using **`vlax-import-type-library`** from inside any AutoLISP expression (that is, always call it from a top-level position).
- In your AutoLISP source file, code the **`vlax-import-type-library`** call before any code that uses method or property wrappers or constants defined in the type library.

See Also

The vlax-typeinfo-available-p function.

vlax-invoke-method

Calls the specified ActiveX method

`(vlax-invoke-method` ***`obj method arg [arg...]`*** **`)`**

This function was known as **`vlax-invoke`** prior to AutoCAD 2000.

Arguments

obj	A VLA-object.
method	A symbol or string naming the method to be called.

arg	Argument to be passed to the method called. No argument type checking is performed.

Return Values

Depends on the method invoked.

Examples

The following example uses the AddCircle method to draw a circle in the current AutoCAD drawing.

The first argument to AddCircle specifies the location of the center of the circle. The method requires the center to be specified as a variant containing a three-element array of doubles. You can use **`vlax-3d-point`** to convert an AutoLISP point list to the required variant data type:

```
_$ (setq circCenter (vlax-3d-point '(3.0 3.0 0.0)))
#<variant 8197 ...>
```

Now use **`vlax-invoke-method`** to draw a circle with the AddCircle method:

```
_$ (setq mycircle (vlax-invoke-method mspace 'AddCircle circCenter
3.0))
#<VLA-OBJECT IAcadCircle 00bfd6e4>
```

See Also

The vlax-get-property, vlax-method-applicable-p, vlax-property-available-p, and vlax-put-property functions.

vlax-ldata-delete

Erases LISP data from a drawing dictionary

`(vlax-ldata-delete` ***`dict key [private]`*** **`)`**

Arguments

dict	A VLA-object, AutoCAD drawing entity object, or a string naming a global dictionary.
key	A string specifying the dictionary key.
private	If a non-`nil` value is specified for *private* and **`vlax-ldata-delete`** is called from a separate-namespace VLX, **`vlax-ldata-delete`** deletes private LISP data from *dict*. (See vlax-ldata-get for examples using this argument.)

Return Values

`T`, if successful, otherwise `nil` (for example, the data did not exist).

Examples

Add LISP data to a dictionary:

```
_$ (vlax-ldata-put "dict" "key" '(1))
(1)
```

Use **`vlax-ldata-delete`** to delete the LISP data:

```
_$ (vlax-ldata-delete "dict" "key")
T
```

If **`vlax-ldata-delete`** is called again to remove the same data, it returns `nil` because the data does not exist in the dictionary:

```
_$ (vlax-ldata-delete "dict" "key")
nil
```

See Also

The vlax-ldata-get, vlax-ldata-list, and vlax-ldata-put functions.

vlax-ldata-get

Retrieves LISP data from a drawing dictionary or an object

`(vlax-ldata-get` ***`dict key [default-data] [private]`*** **`)`**

Arguments

dict	A VLA-object, AutoCAD drawing entity object, or a string naming a global dictionary.
key	A string specifying the dictionary key.
default-data	LISP data to be returned if no matching key exists in the dictionary.
private	If a non-`nil` value is specified for *private* and **`vlax-ldata-get`** is called from a separate-namespace VLX, **`vlax-ldata-get`** retrieves private LISP data from *dict*. If you specify *private*, you must also specify *default-data*; you can use `nil` for *default-data*.

Note that a separate-namespace VLX can store both private and non-private data using the same *dict* and *key*. The private data can only be accessed by the same VLX, but any application can retrieve the non-private data.

Return Values

The value of the *key* item.

Examples

Enter the following commands at the Visual LISP Console window:

```
_$ (vlax-ldata-put "mydict" "mykey" "Mumbo Dumbo")
"Mumbo Dumbo"

_$ (vlax-ldata-get "mydict" "mykey")
"Mumbo Dumbo"
```

To test the use of private data from a VLX

1 Enter the following commands at the Visual LISP Console window:

```
_$ (vlax-ldata-put "mydict" "mykey" "Mumbo Dumbo")
"Mumbo Dumbo"

_$ (vlax-ldata-get "mydict" "mykey")
"Mumbo Dumbo"
```

2 Enter the following code in a file and use Make Application to build a VLX from the file. Use the Expert mode of the Make Application Wizard, and select the Separate Namespace option in the Compile Options tab.

```
(vl-doc-export 'ldataput)
(vl-doc-export 'ldataget)
(vl-doc-export 'ldataget-nilt)
(defun ldataput ()
  (princ "This is a test of putting private ldata ")
  (vlax-ldata-put "mydict" "mykey" "Mine! Mine! " T)
)
(defun ldataget ()
  (vlax-ldata-get "mydict" "mykey")
)
(defun ldataget-nilt ()
  (vlax-ldata-get "mydict" "mykey" nil T)
)
```

3 Load the VLX file.

4 Run **`ldataput`** to save private data:

```
_$ (ldataput)
This is a test of putting private ldata
```

Refer back to the code defining **`ldataput`**: this function stores a string containing “Mine! Mine!”

5 Run **ldataget** to retrieve LISP data:

```
_$ (ldataget)
"Mumbo Dumbo"
```

Notice that the data returned by **ldataget** is not the data stored by **ldataput**. This is because **ldataget** does not specify the *private* argument in its call to **vlax-ldata-get**. So the data retrieved by**ldataget** is the data set by issuing **vlax-ldata-put** from the Visual LISP Console in step 1.

```
_$ (ldataget-nilt)
"Mine! Mine! "
```

6 Run **ldataget-nilt** to retrieve LISP data:

```
_$ (ldataget-nilt)
"Mine! Mine! "
```

This time the private data saved by **ldataput** is returned, because **ldataget-nilt** specifies the *private* argument in its call to **vlax-ldata-get**.

7 From the Visual LISP Console prompt, issue the same call that **ldataget-nilt** uses to retrieve private data:

```
_$ (vlax-ldata-get "mydict" "mykey" nil T)
"Mumbo Dumbo"
```

The *private* argument is ignored when **vlax-ldata-get** is issued outside of a separate-namespace VLX. If non-private data exists for the specified *dict* and *key* (as in this instance), that data will be retrieved.

See Also

The vlax-ldata-put, vlax-ldata-delete, and vlax-ldata-list functions.

vlax-ldata-list

Lists LISP data in a drawing dictionary

(vlax-ldata-list *dict [private]*)

Arguments

dict	A VLA-object, AutoCAD drawing entity object, or a string naming a global dictionary.
private	If **vlax-ldata-list** is called from a separate-namespace VLX and a non-`nil` value is specified for *private*, **vlax-ldata-list** retrieves only private data stored by the same VLX. (See vlax-ldata-get for examples using this argument.)

Return Values

An associative list consisting of pairs (key . value).

Examples

Use **`vlax-ldata-put`** to store LISP data in a dictionary:

```
_$ (vlax-ldata-put "dict" "cay" "Mumbo Jumbo ")
"Mumbo Jumbo "
```

```
_$ (vlax-ldata-put "dict" "say" "Floobar ")
"Floobar "
```

Use **`vlax-ldata-list`** to display the LISP data stored in “dict”:

```
_$ (vlax-ldata-list "dict")
(("say" . "Floobar ") ("cay" . "Mumbo Jumbo "))
```

See Also

The vlax-ldata-get, vlax-ldata-delete, and vlax-ldata-put functions.

vlax-ldata-put

Stores LISP data in a drawing dictionary or an object

`(vlax-ldata-put` ***`dict key data [private]`*** **`)`**

Arguments

dict	A VLA-object, AutoCAD drawing entity object, or a string naming a global dictionary.
key	A string specifying the dictionary key.
data	LISP data to be stored in the dictionary.
private	If **`vlax-ldata-put`** is called from a separate-namespace VLX and a non-`nil` value is specified for *private*, **`vlax-ldata-put`** marks the data as retrievable only by the same VLX.

Return Values

The value of *data*.

Examples

```
_$ (vlax-ldata-put "dict" "key" '(1))
(1)
```

```
_$ (vlax-ldata-put "dict" "cay" "Gumbo jumbo")
"Gumbo jumbo"
```

See Also

The vlax-ldata-get, vlax-ldata-delete, and vlax-ldata-list functions.

vlax-ldata-test

Determines if data can be saved over a session boundary

```
(vlax-ldata-test data)
```

Arguments

data	Any LISP data to be tested.

Return Values

`T`, if the data can be saved and restored over the session boundary, `nil` otherwise.

Examples

Determine if a string can be saved as ldata over a session boundary:

```
_$ (vlax-ldata-test "Gumbo jumbo")
T
```

Determine if a function can be saved as ldata over a session boundary:

```
_$ (vlax-ldata-test yinyang)
nil
```

See Also

The vlax-ldata-get, vlax-ldata-delete, and vlax-ldata-list, and vlax-ldata-put functions.

vlax-make-safearray

Creates a safearray

```
(vlax-make-safearray type '(l-bound . u-bound)
  ['(l-bound . u-bound)...)]
```

A maximum of 16 dimensions can be defined for an array. The elements in the array are initialized as follows:

Numbers	0
Strings	Zero-length string.
Booleans	`:vlax-false`
Object	`nil`
Variant	Uninitialized (vlax-vbEmpty)

Arguments

type	The type of safearray. Specify one of the following constants: **vlax-vbInteger** (2) Integer **vlax-vbLong** (3) Long integer **vlax-vbSingle** (4) Single-precision floating-point number **vlax-vbDouble** (5) Double-precision floating-point number **vlax-vbString** (8) String **vlax-vbObject** (9) Object **vlax-vbBoolean** (11) Boolean **vlax-vbVariant** (12) Variant The integer shown in parentheses indicates the value to which the constant evaluates. It is recommended that you specify the constant in your argument, not the integer value, in case the value changes in later releases of AutoCAD.
'(l-bound . u-bound)	Lower and upper index boundaries of a dimension.

Return Values

The safearray created.

Examples

Create a single-dimension safearray consisting of doubles, beginning with index 0:

```
_$ (setq point (vlax-make-safearray vlax-vbDouble '(0 . 3)))
#<safearray...>
```

Use the **vlax-safearray->list** function to display the contents of the safearray as a list:

```
_$ (vlax-safearray->list point)
(0.0 0.0 0.0 0.0)
```

The result shows each element of the array was initialized to zero.

Create a two-dimension array of strings, with each dimension starting at index 1:

```
_$ (setq matrix (vlax-make-safearray vlax-vbString '(1 . 2) '(1 .
2) ))
#<safearray...>
```

See Also

The vlax-make-variant, vlax-safearray-fill, vlax-safearray-get-dim, vlax-safearray-get-element, vlax-safearray-get-l-bound, vlax-safearray-get-u-bound, vlax-safearray-put-element, vlax-safearray-type, vlax-safearray->list, and vlax-variant-value functions. For more information on using these functions, see “Working with Safearrays” in the *Visual LISP Developer’s Guide*.

vlax-make-variant

Creates a variant data type

(vlax-make-variant *[value] [type]*)

Arguments

value	The value to be assigned to the variant. If omitted, the variant is created with the vlax-vbEmpty type (uninitialized).
type	The type of variant. This can be represented by one of the following constants:

vlax-vbEmpty (0)	Uninitialized (default value)
vlax-vbNull (1)	Contains no valid data

vlax-vbInteger (2) Integer

vlax-vbLong (3) Long integer

vlax-vbSingle (4) Single-precision floating-point number

vlax-vbDouble (5) Double-precision floating-point number

vlax-vbString (8) String

vlax-vbObject (9) Object

vlax-vbBoolean (11) Boolean

vlax-vbArray (8192) Array

The integer shown in parentheses indicates the value to which the constant evaluates. It is recommended that you specify the constant in your argument, not the integer value, because the value may change in later releases of AutoCAD.

If you do not specify a *type*, **`vlax-make-variant`** assigns a default data type based on the data type of the *value* it receives. The following list identifies the default variant data type assigned to each LISP data type:

nil `vlax-vbEmpty`

:vlax-null `vlax-vbNull`

integer `vlax-vbLong`

real `vlax-vbDouble`

string `vlax-vbString`

VLA-object `vlax-vbObject`

:vlax-true, **:vlax-false** `vlax-vbBoolean`

variant Same as the type of initial value

vlax-make-safearray `vlax-vbArray`

Return Values

The variant created.

Examples

Create a variant using the defaults for **`vlax-make-variant`**:

```
_$ (setq varnil (vlax-make-variant))
#<variant 0 >
```

The function creates an uninitialized (vlax-vbEmpty) variant by default. You can accomplish the same thing explicitly with the following call:

```
_$ (setq varnil (vlax-make-variant nil))
#<variant 0 >
```

Create an integer variant and set its value to 5:

```
_$ (setq varint (vlax-make-variant 5 vlax-vbInteger))
#<variant 2 5>
```

Repeat the previous command, but omit the *type* argument and see what happens:

```
_$ (setq varint (vlax-make-variant 5))
#<variant 3 5>
```

By default, **`vlax-make-variant`** assigned the specified integer value to a Long integer data type, not Integer, as you might expect. This highlights the importance of explicitly stating the type of variant you want when working with numbers.

Omitting the *type* argument for a string produces predictable results:

```
_$ (setq varstr (vlax-make-variant "ghost"))
#<variant 8 ghost>
```

To create a variant containing arrays, you must specify type `vlax-vbArray`, along with the type of data in the array. For example, to create a variant containing an array of doubles, first set a variable's value to an array of doubles:

```
_$ (setq 4dubs (vlax-make-safearray vlax-vbDouble '(0 . 3)))
#<safearray...>
```

Then take the array of doubles and assign it to a variant:

```
_$ (vlax-make-variant 4dubs)
#<variant 8197 ...>
```

See Also

The vlax-make-safearray, vlax-variant-change-type, vlax-variant-type, and vlax-variant-value functions. For more information on using variants, see "Working with Variants" in the *Visual LISP Developer's Guide*.

vlax-map-collection

Applies a function to all objects in a collection

(vlax-map-collection *obj function*)

Arguments

obj	A VLA-object representing a collection.
function	A symbol or lambda expression to be applied to *obj*.

Return Values

The *obj* first argument.

Examples

```
(vlax-map-collection (vla-get-ModelSpace acadDocument) 'vlax-dump-
object)
; IAcadLWPolyline: AutoCAD Lightweight Polyline Interface
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00a4ae24>
;   Area (RO) = 2.46556
;   Closed = 0
;   Color = 256
;   ConstantWidth = 0.0
;   Coordinate = ...Indexed contents not shown...
;  Coordinates = (8.49917 7.00155 11.2996 3.73137 14.8 5.74379 ... )
;   Database (RO) = #<VLA-OBJECT IAcadDatabase 01e3da44>
;   Elevation = 0.0
;   Handle (RO) = "53"
;   HasExtensionDictionary (RO) = 0
;   Hyperlinks (RO) = #<VLA-OBJECT IAcadHyperlinks 01e3d7d4>
;   Layer = "0"
;   Linetype = "BYLAYER"
;   LinetypeGeneration = 0
;   LinetypeScale = 1.0
;   Lineweight = -1
;   Normal = (0.0 0.0 1.0)
;   ObjectID (RO) = 28895576
;   ObjectName (RO) = "AcDbPolyline"
;   PlotStyleName = "ByLayer"
;   Thickness = 0.0
;   Visible = -1
T
```

vlax-method-applicable-p

Determines if an object supports a particular method

(vlax-method-applicable-p *obj method*)

Arguments

obj	A VLA-object.
method	A symbol or string containing the name of the method to be checked.

Return Values

`T`, if the object supports the method, `nil` otherwise.

Examples

The following commands are issued against a LightweightPolyline object:

```
_$ (vlax-method-applicable-p WhatsMyLine 'copy)
T
```

```
_$ (vlax-method-applicable-p WhatsMyLine 'AddBox)
nil
```

See Also

The vlax-property-available-p function.

vlax-object-released-p

Determines if an object has been released

(vlax-object-released-p *obj*)

NOTE Erasing a VLA-object (using **`command`** ERASE or **`vla-erase`**) does not release the object. A VLA-object is not released until you either invoke **`vlax-release-object`** on the object, normal AutoLISP garbage collection occurs, or the drawing database is destroyed at the end of the drawing session.

Arguments

obj	A VLA-object.

Return Values

`T`, if the object is released (no AutoCAD drawing object is attached to *obj*), `nil`, if the object has not been released.

Examples

Attach an Excel application to the current AutoCAD drawing:

```
_$ (setq excelobj (vlax-get-object "Excel.Application"))
#<VLA-OBJECT _Application 00168a54>
```

Release the Excel object:

```
_$ (vlax-release-object excelobj)
1
```

Issue **`vlax-object-released-p`** to verify the object was released:

```
_$ (vlax-object-released-p excelobj)
T
```

vlax-product-key

Returns the AutoCAD Window registry path

`(vlax-product-key)`

The AutoCAD registry path can be used to register an application for demand loading.

Return Values

A string containing the AutoCAD registry path.

Examples

```
_$ (vlax-product-key)
"Software\\Autodesk\\AutoCAD\\R15.0\\ACAD-1:409"
```

vlax-property-available-p

Determines if an object has a specified property

(vlax-property-available-p *obj prop [check-modify]*)

Arguments

obj	A VLA-object.
property	A symbol or string naming the property to be checked.
check-modify	If `T` is specified for this argument, **`vlax-property-available-p`** also checks that the property can be modified.

Return Values

`T`, if the object has the specified property, otherwise `nil`. If `T` is specified for the *check-modify* argument, **`vlax-property-available-p`** returns `nil` if either the property is not available *or* the property cannot be modified.

Examples

The following examples apply to a LightweightPolyline object:

```
_$ (vlax-property-available-p WhatsMyLine 'Color)
T
```

```
_$ (vlax-property-available-p WhatsMyLine 'center)
nil
```

The following examples apply to a Circle object:

```
_$ (vlax-property-available-p myCircle 'area)
T
```

Note how supplying the optional third argument changes the result:

```
_$ (vlax-property-available-p myCircle 'area T)
nil
```

The function returns `nil` because, although the circle has an “area” property, that property cannot be modified.

See Also

The vlax-method-applicable-p and vlax-put-property functions.

vlax-put-property

Set the property of an ActiveX object

(vlax-put-property *obj property arg*)

This function was formerly known as vlax-put.

Arguments

obj	A VLA-object.
property	A symbol or string naming the property to be set.
arg	The value to be set.

Return Values

`Nil`, if successful.

Examples

Color an object red:

```
_$ (vlax-put-property vlaobj 'Color 1)
nil
```

See Also

The vlax-get-property and vlax-property-available-p functions.

vlax-read-enabled-p

Determines if an object can be read

(vlax-read-enabled-p *obj*)

Arguments

obj	A VLA-object.

Return Values

`T`, if the object is readable, otherwise `nil`.

vlax-release-object

Releases a drawing object

(vlax-release-object *obj*)

Arguments

obj	A VLA-object.

After release, the drawing object is no longer accessible through *obj*.

Return Values

Unspecified.

vlax-remove-cmd

Removes a single command or a command group

(vlax-remove-cmd *global-name*)

Removes a single command or the whole command group for the current AutoCAD session.

Arguments

global-name	Either a string naming the command, or `T`. If *global-name* is `T`, the whole command group VLC-*AppName* (for example, `VLC-VLIDE`) is deleted.

Return Values

`T`, if successful, `nil` otherwise (for example, the command is not defined).

Examples

Remove a command defined with **`vlax-add-cmd`**:

```
_$ (vlax-remove-cmd "hello-autocad")
T
```

Repeat the **`vlax-remove-cmd`**:

```
_$ (vlax-remove-cmd "hello-autocad")
nil
```

This time **`vlax-remove-cmd`** returns `nil`, because the specified command does not exist anymore.

See Also

The vlax-add-cmd function.

vlax-safearray-fill

Stores data in the elements of a safearray

`(vlax-safearray-fill` ***`var`*** **`'`*`element-values`*`)`**

Arguments

var	A variable whose data type is a safearray.
'element-values	A list of values to be stored in the array. You can specify as many values as there are elements in the array. If you specify fewer values than there are elements, the remaining elements retain their current value. For multi-dimension arrays, *element-values* must be a list of lists, with each list corresponding to a dimension of the array.

Return Values

var

Examples

Create a single-dimension array of doubles:

```
_$ (setq sa (vlax-make-safearray vlax-vbdouble '(0 . 2)))
#<safearray...>
```

Use **`vlax-safearray-fill`** to populate the array:

```
_$ (vlax-safearray-fill sa '(1 2 3))
#<safearray...>
```

List the contents of the array:

```
_$ (vlax-safearray->list sa)
(1.0 2.0 3.0)
```

Use **`vlax-safearray-fill`** to set the first element in the array:

```
_$ (vlax-safearray-fill sa '(-66))
#<safearray...>
```

List the contents of the array:

```
_$ (vlax-safearray->list sa)
(-66.0 2.0 3.0)
```

Notice that only the first element in the array has been changed; the remaining elements are unaffected and retain the value you previously set them to. If you need to change the second or third elements and leave the first element unaffected, use **vlax-put-element**.

Instruct **vlax-safearray-fill** to set four elements in an array that contains only three elements:

```
_$ (vlax-safearray-fill sa '(1 2 3 4))
Error: Assertion failed: safearray-fill failed. Too many elements.
```

The **vlax-safearray-fill** function returns an error if you specify more elements than the array contains.

To assign values to a multi-dimensional array, specify a list of lists to **vlax-safearray-fill**, with each list corresponding to a dimension. The following command creates a two-dimension array of strings containing three elements in each dimension:

```
_$ (setq mat2 (vlax-make-safearray vlax-vbString '(0 . 1) '(1 . 3)))
#<safearray...>
```

Use **vlax-safearray-fill** to populate the array:

```
_$ (vlax-safearray-fill mat2 '(("a" "b" "c") ("d" "e" "f")))
#<safearray...>
```

Call the **vlax-safearray->list** function to confirm the contents of `mat2`:

```
_$ (vlax-safearray->list mat2)
(("a" "b" "c") ("d" "e" "f"))
```

See Also

The vlax-make-safearray, vlax-safearray-get-dim, vlax-safearray-get-element, vlax-safearray-get-l-bound, vlax-safearray-get-u-bound, vlax-safearray-put-element, vlax-safearray-type, vlax-safearray->list, and vlax-variant-value functions.

vlax-safearray-get-dim

Returns the number of dimensions in a safearray object

(vlax-safearray-get-dim *var*)

Arguments

var	A variable whose data type is a safearray.

Return Values

An integer identifying the number of dimensions in *var*. An error occurs if *var* is not a safearray.

Examples

Set `sa-int` to a single-dimension safearray with one dimension:

```
_$ (setq sa-int (vlax-make-safearray vlax-vbinteger '(1 . 4)))
#<safearray...>
```

Use **`vlax-safearray-get-dim`** to return the number of dimensions in `sa-int`:

```
_$ (vlax-safearray-get-dim sa-int)
1
```

See Also

The vlax-make-safearray, vlax-safearray-get-l-bound, and vlax-safearray-get-u-bound functions.

vlax-safearray-get-element

Returns an element from an array

`(vlax-safearray-get-element` ***`var element`*** **`...)`**

Arguments

var	A variable whose data type is a safearray.
element...	Integers specifying the indexes of the element to be retrieved. For an array with one dimension, specify a single integer. For multi-dimension arrays, specify as many indexes as there are dimensions.

Return Values

The value of the element.

Examples

Create an array with two dimensions, each dimension starting at index 1:

```
_$ (setq matrix (vlax-make-safearray vlax-vbString '(1 . 2) '(1 .
2) ))
#<safearray...>
```

Use **`vlax-safearray-put-element`** to populate the array:

```
_$ (vlax-safearray-put-element matrix 1 1 "a")
"a"

_$ (vlax-safearray-put-element matrix 1 2 "b")
"b"

_$ (vlax-safearray-put-element matrix 2 1 "c")
"c"

_$ (vlax-safearray-put-element matrix 2 2 "d")
"d"
```

Use **`vlax-safearray-get-element`** to retrieve the second element in the first dimension of the array:

```
_$ (vlax-safearray-get-element matrix 1 2)
"b"
```

See Also

The vlax-make-safearray, vlax-safearray-get-dim, vlax-safearray-get-l-bound, vlax-safearray-get-u-bound, and vlax-safearray-put-element functions.

vlax-safearray-get-l-bound

Returns the lower boundary (starting index) of a dimension of an array

`(vlax-safearray-get-l-bound` ***`var dim`*****`)`**

Arguments

var	A variable whose data type is a safearray.
dim	A dimension of the array. The first dimension is dimension 1.

Return Values

An integer representing the lower boundary (starting index) of the dimension. If *var* is not an array, or *dim* is invalid (for example, 0, or a number greater than the number of dimensions in the array), an error results.

Examples

The following examples evaluate a safearray defined as follows:

```
(vlax-make-safearray vlax-vbString '(1 . 2) '(0 . 1) ))
```

Get the starting index value of the array's first dimension:

```
_$ (vlax-safearray-get-l-bound tmatrix 1)
1
```

The first dimension starts with index 1.

Get the starting index value of the second dimension of the array:

```
_$ (vlax-safearray-get-l-bound tmatrix 2)
0
```

The second dimension starts with index 0.

See Also

The vlax-make-safearray, vlax-safearray-get-dim, and vlax-safearray-get-u-bound functions.

vlax-safearray-get-u-bound

Returns the upper boundary (end index) of a dimension of an array

(vlax-safearray-get-u-bound *var dim*)

Arguments

var	A variable whose data type is a safearray.
dim	A dimension of the array. The first dimension is dimension 1.

Return Values

An integer representing the upper boundary (end index) of the dimension. If *var* is not an array, or *dim* is invalid (for example, 0, or a number greater than the number of dimensions in the array), an error results.

Examples

The following examples evaluate a safearray defined as follows:

```
(vlax-make-safearray vlax-vbString '(1 . 2) '(0 . 1) ))
```

Get the end index value of the array's first dimension:

```
_$ (vlax-safearray-get-u-bound tmatrix 1)
2
```

The first dimension ends with index 2.

Get the end index value of the second dimension of the array:

```
_$ (vlax-safearray-get-u-bound tmatrix 2)
1
```

The second dimension starts with index 1.

See Also

The vlax-make-safearray, vlax-safearray-get-dim, and vlax-safearray-get-l-bound functions.

vlax-safearray-put-element

Adds an element to an array

(vlax-safearray-put-element *var index... value*)

Arguments

var	A variable whose data type is a safearray.
index...	A set of index values pointing to the element you are assigning a value to. For a single-dimension array, specify one index value; for a two-dimension array, specify two index values, and so on.
value	The value to assign the safearray element.

Return Values

The *value* assigned to the element.

Examples

Create a single-dimension array consisting of doubles:

```
_$ (setq point (vlax-make-safearray vlax-vbDouble '(0 . 2)))
#<safearray...>
```

Use **vlax-safearray-put-element** to populate the array:

```
_$ (vlax-safearray-put-element point 0 100)
100
```

```
_$ (vlax-safearray-put-element point 1 100)
100
```

```
_$ (vlax-safearray-put-element point 2 0)
0
```

Create a two-dimension array consisting of strings:

```
_$ (setq matrix (vlax-make-safearray vlax-vbString '(1 . 2) '(1 .
2) ))
#<safearray...>
```

Use **`vlax-safearray-put-element`** to populate the array:

```
_$ (vlax-safearray-put-element matrix 1 1 "a")
"a"
```

```
_$ (vlax-safearray-put-element matrix 1 2 "b")
"b"
```

```
_$ (vlax-safearray-put-element matrix 2 1 "c")
"c"
```

```
_$ (vlax-safearray-put-element matrix 2 2 "d")
"d"
```

Note that you can also populate arrays using the **`vlax-safearray-fill`** function. The following function call accomplishes the same task as three **`vlax-safearray-put-element`** calls:

```
(vlax-safearray-fill matrix '(("a" "b") ("c" "d")))
```

See Also

The vlax-safearray-get-element, vlax-safearray-fill, and vlax-safearray-type functions.

vlax-safearray-type

Returns the data type of a safearray

`(vlax-safearray-type` ***`var`*****`)`**

Arguments

var	A variable containing a safearray.

Return Values

If *var* contains a safearray, one of the following integers is returned:

2	Integer (`vlax-vbInteger`)

3	Long integer (`vlax-vbLong`)
4	Single-precision floating-point number (`vlax-vbSingle`)
5	Double-precision floating-point number (`vlax-vbDouble`)
8	String (`vlax-vbString`)
9	Object (`vlax-vbObject`)
11	Boolean (`vlax-vbBoolean`)
12	Variant (`vlax-vbVariant`)

If *var* does not contain a safearray, an error results.

Examples

Create a single-dimension array of doubles and a two-dimension array of strings:

```
_$ (setq point (vlax-make-safearray vlax-vbDouble '(0 . 2)))
#<safearray...>
```

```
_$ (setq matrix (vlax-make-safearray vlax-vbString '(1 . 2) '(1 .
2) ))
#<safearray...>
```

Use **`vlax-safearray-type`** to verify the data type of the safearrays:

```
_$ (vlax-safearray-type point)
5
```

```
_$ (vlax-safearray-type matrix)
8
```

See Also

The vlax-make-safearray function.

vlax-safearray->list

Returns the elements of a safearray in list form

`(vlax-safearray->list` ***`var`*****`)`**

Arguments

var	A variable containing a safearray.

Return Values

A list.

Examples

Create a single-dimension array of doubles:

```
_$ (setq point (vlax-make-safearray vlax-vbDouble '(0 . 2)))
#<safearray...>
```

Use **`vlax-safearray-put-element`** to populate the array:

```
_$ (vlax-safearray-put-element point 0 100)
100

_$ (vlax-safearray-put-element point 1 100)
100

_$ (vlax-safearray-put-element point 2 0)
0
```

Convert the array to a list:

```
_$ (setq pointlist (vlax-safearray->list point))
(100.0 100.0 0.0)
```

The following example demonstrates how a two-dimension array of strings is displayed by **`vlax-safearray->list`**:

```
_$ (vlax-safearray->list matrix)
(("a" "b") ("c" "d"))
```

See Also

The vlax-make-safearray, vlax-safearray-fill, and vlax-safearray-put-element functions.

vlax-tmatrix

Returns a suitable representation for a 4 × 4 transformation matrix to be used in VLA methods

`(vlax-tmatrix` ***`list`*** **`)`**

Arguments

list	A list of four lists, each containing four numbers, representing transformation matrix elements.

Return Values

A variant of type safearray, representing the 4 × 4 transformation matrix.

Examples

Define a transformation matrix and assign its value to variable `tmatrix`:

```
_$ (setq tmatrix (vlax-tmatrix '((1 1 1 0) (1 2 3 0) (2 3 4 5) (2 9
8 3))))
#<variant 8197 ...>
```

Use **`vlax-safearray->list`** to view the value of `tmatrix` in list form:

```
_$ (vlax-safearray->list (vlax-variant-value tmatrix))
((1.0 1.0 1.0 0.0) (1.0 2.0 3.0 0.0) (2.0 3.0 4.0 5.0) (2.0 9.0 8.0
3.0))
```

The following code example creates a line and rotates it 90 degrees using a transformation matrix:

```
(defun Example_TransformBy () ; / lineObj startPt endPt matList
transMat)

(vl-load-com)      ; Load ActiveX support
(setq acadObject   (vlax-get-acad-object))
(setq acadDocument (vla-get-ActiveDocument acadObject))
(setq mSpace       (vla-get-ModelSpace acadDocument))

;; Create a line

 (setq startPt (getpoint "Pick the start point"))
 (setq endPt (vlax-3d-point (getpoint startPt "Pick the end
point")))
 (setq lineObj (vla-addline mSpace (vlax-3d-point startPt) endPt))

;;; Initialize the transMat variable with a transformation matrix
;;; that will rotate an object by 90 degrees about the point(0,0,0).
;;; Begin by Creating a list of four lists, each containing four
;;; numbers, representing transformation matrix elements.

 (setq matList (list '(0 -1 0 0) '(1 0 0 0) '(0 0 1 0) '(0 0 0 1)))

;;; Use vlax-tmatrix to convert the list to a variant.

 (setq transmat (vlax-tmatrix matlist))

;;;  Transform the line using the defined transformation matrix

 (vla-transformby lineObj transMat)
 (vla-zoomall acadObject)

 (princ "The line is transformed ")
 (princ)
)
```

vlax-typeinfo-available-p

Determines whether TypeLib information is present for the specified type of object

Visual LISP requires TypeLib information to determine whether a method or property is available for an object. Some objects may not have TypeLib information (for example, AcadDocument).

`(vlax-typeinfo-available-p` *`obj`*`)`

Arguments

obj	A VLA-object.

Return Values

`T`, if TypeLib information is available, otherwise `nil`.

See Also

The vlax-import-type-library function.

vlax-variant-change-type

Returns the value of a variant after changing it from one data type to another

`(vlax-variant-change-type` *`var type`*`)`

The **`vlax-variant-change-type`** function returns the value of the specified variable after converting that value to the specified variant type.

Arguments

var	A variable whose value is a variant.
type	The type of variant to return, using the value of *var* (the value of *var* is unchanged). The *type* value can be represented by one of the following constants:

vlax-vbEmpty (0) Uninitialized

vlax-vbNull (1) Contains no valid data

vlax-vbInteger (2) Integer

vlax-vbLong (3) Long integer

vlax-vbSingle (4) Single-precision floating-point number

vlax-vbDouble (5) Double-precision floating-point number

vlax-vbString (8) String

vlax-vbObject (9) Object

vlax-vbBoolean (11) Boolean

vlax-vbArray (8192) Array

The integer shown in parentheses indicates the value to which the constant evaluates. It is recommended that you specify the constant in your argument, not the integer value, in case the value changes in later releases of AutoCAD.

Return Values

The value of *var*, after converting it to the specified variant type, or `nil`, if *var* could not be converted to the specified type.

Examples

Set a variable named `varint` to a variant value:

```
_$ (setq varint (vlax-make-variant 5))
#<variant 3 5>
```

Set a variable named `varintstr` to the value contained in `varint`, but convert that value to a string:

```
_$ (setq varintStr (vlax-variant-change-type varint vlax-vbstring))
#<variant 8 5>
```

Check the value of `varintstr`:

```
_$ (vlax-variant-value varintStr)
"5"
```

This confirms that `varintstr` contains a string.

See Also

The vlax-variant-type and vlax-variant-value functions.

vlax-variant-type

Determines the data type of a variant

(vlax-variant-type *var*)

Arguments

var	A variable whose value is a variant.

Return Values

If *var* contains a variant, one of the following integers is returned:

0	Uninitialized (`vlax-vbEmpty`)
1	Contains no valid data (`vlax-vbNull`)
2	Integer (`vlax-vbInteger`)
3	Long integer (`vlax-vbLong`)
4	Single-precision floating-point number (`vlax-vbSingle`)
5	Double-precision floating-point number (`vlax-vbDouble`)
8	String (`vlax-vbString`)
9	Object (`vlax-vbObject`)
11	Boolean (`vlax-vbBoolean`)
8192 + *n*	Safearray (`vlax-vbArray`) of some data type. For example, an array of doubles (`vlax-vbDouble`) returns 8197 (8192 + 5).

If *var* does not contain a variant, an error results.

Examples

Set a variant to `nil` and display the variant's data type:

```
_$ (setq varnil (vlax-make-variant nil))
#<variant 0 >
```

```
_$ (vlax-variant-type varnil)
0
```

Set a variant to an integer value and explicitly define the variant as an integer data type:

```
_$ (setq varint (vlax-make-variant 5 vlax-vbInteger))
#<variant 2 5>

_$ (vlax-variant-type varint)
2
```

Set a variant to an integer value and display the variant's data type:

```
_$ (setq varint (vlax-make-variant 5))
#<variant 3 5>

_$ (vlax-variant-type varint)
3
```

Notice that without explicitly defining the data type to **`vlax-make-variant`**, an integer assignment results in a Long integer data type.

Set a variant to a string and display the variant's data type:

```
_$ (setq varstr (vlax-make-variant "ghost"))
#<variant 8 ghost>

_$ (vlax-variant-type varstr)
8
```

Create a safearray of doubles, assign the safearray to a variant, and display the variant's data type:

```
_$ (setq 4dubs (vlax-make-safearray vlax-vbDouble '(0 . 3)))
#<safearray...>

_$ (setq var4dubs (vlax-make-variant 4dubs))
#<variant 8197 ...>

_$ (vlax-variant-type var4dubs)
8197
```

A variant type value greater than 8192 indicates that the variant contains some type of safearray. Subtract 8192 from the return value to determine the data type of the safearray. In this example, 8197-8192=5 (`vlax-vbDouble`).

Assign a real value to a variable, then issue **`vlax-variant-type`** to check the variable's data type:

```
_$ (setq notvar 6.0)
6.0

_$ (vlax-variant-type notvar)
; *** ERROR: bad argument type: variantp 6.0
```

This last example results in an error, because the variable passed to **`vlax-variant-type`** does not contain a variant.

See Also

The vlax-make-safearray, vlax-make-variant, vlax-variant-change-type, and vlax-variant-value functions.

vlax-variant-value

Returns the value of a variant

(vlax-variant-value *var*)

Arguments

var	A variable whose value is a variant.

Return Values

The value of the variable. If the variable does not contain a variant, an error occurs.

Examples

```
_$ (vlax-variant-value varstr)
"ghost"

_$ (vlax-variant-value varint)
5

_$ (vlax-variant-value notvar)
; *** ERROR: bad argument type: variantp 6.0
```

The last example results in an error, because `notvar` does not contain a variant.

See Also

The vlax-make-safearray and vlax-make-variant functions.

vlax-vla-object->ename

Transforms a VLA-object to an AutoLISP entity

(vlax-vla-object->ename *obj*)

Arguments

obj	A VLA-object.

Return Values

An AutoLISP entity name (ename data type).

Examples

```
_$ (vlax-vla-object->ename vlaobj)
<Entity name: 27e0540>
```

See Also

The vlax-ename->vla-object function.

vlax-write-enabled-p

Determines if an AutoCAD drawing object can be modified

`(vlax-write-enabled-p` ***`obj`*****`)`**

Arguments

obj	A VLA-object or AutoLISP entity object (ename).

Return Values

`T`, if the AutoCAD drawing object can be modified, `nil` if the object cannot be modified.

vlisp-compile

Compiles AutoLISP source code into a FAS file

`(vlisp-compile` `'`***`mode filename [out-filename]`*****`)`**

NOTE The Visual LISP IDE must be open in order for **`vlisp-compile`** to work.

Arguments

mode The compiler mode, which can be one of the following symbols:

`st`	Standard build mode
`lsm`	Optimize and link indirectly
`lsa`	Optimize and link directly

filename	A string identifying the AutoLISP source file. If the source file is in the AutoCAD Support File Search Path, you can omit the path when specifying the file name. If you omit the file extension, *.lsp* is assumed.
out-filename	A string identifying the compiled output file. If you do not specify an output file, **`vlisp-compile`** names the output with the same name as the input file, but replaces the extension with *.fas*. Note that if you specify an output file name but do not specify a path name for either the input or the output file, **`vlisp-compile`** places the output file in the AutoCAD install directory.

Return Values

`T`, if compilation is successful, `nil` otherwise.

Examples

Assuming that *yinyang.lsp* resides in a directory that is in the AutoCAD Support File Search Path, the following command compiles this program:

```
_$ (vlisp-compile 'st "yinyang.lsp")
T
```

The output file is named *yinyang.fas* and resides in the same directory as the source file.

The following command compiles *yinyang.lsp* and names the output file *GoodKarma.fas*:

```
(vlisp-compile 'st "yinyang.lsp" "GoodKarma.fas")
```

Note that the output file from the previous command resides in the AutoCAD install directory, *not* the directory where *yinyang.lsp* resides. The following command compiles *yinyang.lsp* and directs the output file to the *c:\my documents* directory:

```
(vlisp-compile 'st "yinyang.lsp" "c:/my documents/GoodKarma")
```

This last example identifies the full path of the file to be compiled:

```
(vlisp-compile ‘st "c:/program files/autocad 2000i/Sample/
yinyang.lsp")
```

The output file from this command is named *yinyang.fas* and resides in the same directory as the input file.

See Also

The “Compiling a Program from a File” topic in the *Visual LISP Developer’s Guide*.

vlr-acdb-reactor

Constructs a reactor object that notifies when an object is added to, modified in, or erased from a drawing database

The **`vlr-acdb-reactor`** function constructs a database reactor object.

`(vlr-acdb-reactor *data callbacks*)`

Arguments

data	Any AutoLISP data to be associated with a reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form: (*event-name* . *callback_function*) where *event-name* is one of the symbols listed in the Database Reactor Events table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments: *reactor_object* The VLR object that called the callback function *obj* The database object (AutoLISP entity) associated with the event

Database reactor events

Name	Event
:vlr-objectAppended	An object has been appended to the drawing database.
:vlr-objectUnAppended	An object has been detached from the drawing database, e.g., by using UNDO.
:vlr-objectReAppended	A detached object has been restored in the drawing database, e.g., by using REDO.
:vlr-objectOpenedForModify	An object is about to be changed.

Database reactor events (*continued*)

Name	Event
:vlr-objectModified	An object has been changed.
:vlr-objectErased	An object has been flagged as being erased.
:vlr-objectUnErased	An object's erased-flag has been removed.

vlr-add

Enables a disabled reactor object

```
(vlr-add obj)
```

Arguments

obj	A VLR object representing the reactor to be enabled.

Return Values

The *obj* argument.

See Also

The vlr-added-p and vlr-remove functions.

vlr-added-p

Tests to determine if a reactor object is enabled

```
(vlr-added-p obj)
```

Arguments

obj	A VLR object representing the reactor to be tested.

Return Values

`T`, if the specified reactor is enabled, or `nil`, if the reactor is disabled.

See Also

The vlr-add function.

vlr-beep-reaction

Produces a beep sound

```
(vlr-beep-reaction [args])
```

Arguments

This is a predefined callback function that accepts a variable number of arguments, depending on the reactor type. The function can be assigned to an event handler for debugging.

vlr-command-reactor

Constructs an editor reactor that notifies of a command event

```
(vlr-command-reactor data callbacks)
```

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data is to be associated with the reactor.
callbacks	A list of pairs of the following form: (e*vent-name* . *callback_function*) where *event-name* is one of the symbols listed in the Command Reactor Events table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments: *reactor_object* The VLR object that called the callback function *list* A list containing a single element, the string identifying the command.

Return Values

The *reactor_object* argument.

Command reactor events

Event name	Description
:vlr-unknownCommand	A command not known to AutoCAD was issued.
:vlr-commandWillStart	An AutoCAD command has been called.
:vlr-commandEnded	An AutoCAD command has completed.
:vlr-commandCancelled	An AutoCAD command has been canceled.
:vlr-commandFailed	An AutoCAD command failed to complete.

vlr-current-reaction-name

Returns the name (symbol) of the current event, if called from within a reactor's callback

```
(vlr-current-reaction-name)
```

Return Values

A symbol indicating the event that triggered the reactor.

vlr-data

Returns application-specific data associated with a reactor

```
(vlr-data obj)
```

Arguments

obj	A VLR object representing the reactor object from which to extract data.

Return Values

The application-specific data obtained from the reactor object.

Examples

The following example obtains a string associated with the `circleReactor` VLR object:

```
_$ (vlr-data circleReactor)
"Circle Reactor"
```

vlr-data-set

Overwrites application-specific data associated with a reactor

`(vlr-data-set` ***`obj data`*****`)`**

Arguments

obj	A VLR object representing the reactor object whose data is to be overwritten.
data	Any AutoLISP data.

Return Values

The *data* argument.

Examples

Return the application-specific data value attached to a reactor:

```
_$ (vlr-data circleReactor)
"Circle Reactor"
```

Replaces the text string used to identify the reactor:

```
_$ (vlr-data-set circleReactor "Circle Area Reactor")
"Circle Area Reactor"
```

Verify the change:

```
_$ (vlr-data circleReactor)
"Circle Area Reactor"
```

NOTE The **`vlr-data-set`** function should be used with care to avoid creation of circular structures.

vlr-deepclone-reactor

Constructs an editor reactor object that notifies of a deep clone event

```
(vlr-deepclone-reactor data callbacks)
```

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form: (e*vent-name* . *callback_function*) where *event-name* is one of the symbols listed in the "DeepClone reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments: *reactor_object* The VLR object that called the callback function *list* A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "DeepClone reactor callback data."

Return Values

The *reactor_object* argument.

DeepClone reactor events

Event name	Description
:vlr-beginDeepClone	A deep clone operation is beginning.
:vlr-beginDeepCloneXlation	A deep clone operation has two stages. First, each object and any owned objects are cloned. Second, any object ID references are translated to their cloned IDs. This callback occurs between these two stages.
:vlr-abortDeepClone	A deep clone operation is aborting.
:vlr-endDeepClone	A deep clone operation is ending.

DeepClone reactor callback data

Name	List length	Parameters
:vlr-beginDeepClone :vlr-abortDeepClone :vlr-endDeepClone	0	
:vlr-beginDeepCloneXlation	1	An integer containing the return error status; if this value indicates an error, the deep clone operation is terminated.

vlr-docmanager-reactor

Constructs a reactor object that notifies of events relating to drawing-documents

```
(vlr-docmanager-reactor data callbacks)
```

Arguments

data Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.

callbacks A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "DocManager reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "DocManager reactor callback data."

Return Values

The *reactor_object* argument.

DocManager reactor events

Event name	Description
:vlr-documentCreated	A new document was created for a drawing (new or open). Useful for updating your per-document structures.
:vlr-documentToBeDestroyed	A document will be destroyed.
:vlr-documentLockModeWillChange	A command is about to start or finish modifying elements in the document, and is obtaining or releasing a lock on the document.
:vlr-documentLockModeChangeVetoed	A reactor invoked veto on itself from a :vlr-documentLockModeChanged callback.
:vlr-documentLockModeChanged	The lock on the document has been obtained or released.
:vlr-documentBecameCurrent	The current document has been changed. This does not necessarily imply that the document has been activated, because changing the current document is necessary for some operations. To obtain user input, the document must be activated as well.
:vlr-documentToBeActivated	A currently inactive document has just received the activate signal, implying that it is about to become the current document.
:vlr-documentToBeDeactivated	Another window (inside or outside of AutoCAD) has been activated.

DocManager reactor callback data

Name	List length	Parameters
:vlr-documentCreated :vlr-documentToBeDestroyed :vlr-documentBecameCurrent :vlr-documentToBeActivated :vlr-documentToBeDeactivated	1	The affected document object (VLA-object).
:vlr-documentLockModeChangeVetoed	2	First parameter is the affected document object (VLA-object). Second parameter is the global command string passed in for the lock request. If the callback is being made on behalf of an unlock request, the string will be prefixed with "#".
:vlr-documentLockModeWillChange :vlr-documentLockModeChanged	5	First parameter is the affected document object (VLA-object). Second parameter is an integer indicating the lock currently in effect for the document object. Third parameter is an integer indicating the lock mode that will be in effect after the lock is applied. Fourth parameter is the strongest lock mode from all other execution contexts. Fifth parameter is the global command string passed in for the lock request. If the callback is being made on behalf of an unlock request, the string will be prefixed with "#". Lock modes may be any of the following: 1—Auto Write Lock 2—Not Locked 4—Shared Write 8—Read 10—Exclusive Write

vlr-dwg-reactor

Constructs an editor reactor object that notifies of a drawing event (for example, opening or closing a drawing file)

```
(vlr-dwg-reactor data callbacks)
```

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "DWG reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "DWG reactor callback data."

Return Values

The *reactor_object* argument.

DWG reactor events

Event name	Description
:vlr-beginClose	The drawing database is to be closed.
:vlr-databaseConstructed	A drawing database has been constructed.
:vlr-databaseToBeDestroyed	The contents of the drawing database is about to be deleted from memory.
vlr-beginDwgOpen	AutoCAD is about to open a drawing file.

DWG reactor events (*continued*)

Event name	Description
:vlr-endDwgOpen	AutoCAD has ended the open operation.
:vlr-dwgFileOpened	A new drawing has been loaded into the AutoCAD drawing window.
vlr-beginSave	AutoCAD is about to save the drawing file.
vlr-saveComplete	AutoCAD has saved the current drawing to disk.

DWG reactor callback data

Name	List length	Parameters
:vlr-beginClose :vlr-databaseConstructed :vlr-databaseToBeDestroyed	0	
:vlr-beginDwgOpen :vlr-endDwgOpen :vlr-dwgFileOpened	1	A string identifying the file to open.
:vlr-beginSave	1	A string containing the default file name for save; may be changed by the user
:vlr-saveComplete	1	A string containing the actual file name used for the save.

vlr-dxf-reactor

Constructs an editor reactor object that notifies of an event related to reading or writing a DXF file

(vlr-dxf-reactor *data callbacks*)

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form: (*event-name* . *callback_function*)

where *event-name* is one of the symbols listed in the “DXF reactor events” table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table “DXF reactor callback data.”

Return Values

The *reactor_object* argument.

DXF reactor events

Event name	Description
:vlr-beginDxfIn	The contents of a DXF file is to be appended to the drawing database.
:vlr-abortDxfIn	The DXF import was not successful.
:vlr-dxfInComplete	The DXF import was successful.
:vlr-beginDxfOut	AutoCAD is about to export the drawing database into a DXF file.
:vlr-abortDxfOut	The DXF export operation failed.
:vlr-dxfOutComplete	The DXF export operation was successful.

DXF reactor callback data

Name	List length	Parameters
:vlr-beginDxfIn :vlr-abortDxfIn :vlr-dxfInComplete, :vlr-beginDxfOut :vlr-abortDxfOut :vlr-dxfOutComplete	0	

vlr-editor-reactor

Constructs an editor reactor object

(vlr-editor-reactor *data callbacks*)

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "Editor reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function.

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "Editor reactor callback data" on page 325.

Return Values

The *reactor_object* argument.

Editor reactor events

Event name	Description
:vlr-beginClose	The drawing database is to be closed.
:vlr-beginDxfIn	The contents of a DXF file is to be appended to the drawing database.
:vlr-abortDxfIn	The DXF import was not successful.
:vlr-dxfInComplete	The DXF import completed successfully.

Editor reactor events (*continued*)

Event name	Description
:vlr-beginDxfOut	AutoCAD is about to export the drawing database into a DXF file.
:vlr-abortDxfOut	DXF export operation failed.
:vlr-dxfOutComplete	DXF export operation completed successfully.
:vlr-databaseToBeDestroyed	The contents of the drawing database is about to be deleted from memory.
:vlr-unknownCommand	A command not known to AutoCAD was issued.
:vlr-commandWillStart	An AutoCAD command has been called.
vlr-commandEnded	An AutoCAD command has completed.
:vlr-commandCancelled	An AutoCAD command has been canceled.
:vlr-commandFailed	An AutoCAD command failed to complete.
:vlr-lispWillStart	An AutoLISP expression is to be evaluated.
:vlr-lispEnded	Evaluation of an AutoLISP expression has completed.
:vlr-lispCancelled	Evaluation of an AutoLISP expression has been canceled.
:vlr-beginDwgOpen	AutoCAD is about to open a drawing file.
:vlr-endDwgOpen	AutoCAD has ended the open operation.
:vlr-dwgFileOpened	A new drawing has been loaded into the AutoCAD drawing window.
:vlr-beginSave	AutoCAD is about to save the drawing file.
:vlr-saveComplete	AutoCAD has saved the current drawing to disk.
:vlr-sysVarWillChange	AutoCAD is about to change the value of a system variable.
:vlr-sysVarChanged	The value of a system variable has changed.

Editor reactor events (*continued*)

Event name	Description
:vlr-beginDxfOut	AutoCAD is about to export the drawing database into a DXF file.
:vlr-abortDxfOut	DXF export operation failed.
:vlr-dxfOutComplete	DXF export operation completed successfully.
:vlr-databaseToBeDestroyed	The contents of the drawing database is about to be deleted from memory.
:vlr-unknownCommand	A command not known to AutoCAD was issued.
:vlr-commandWillStart	An AutoCAD command has been called.
vlr-commandEnded	An AutoCAD command has completed.
:vlr-commandCancelled	An AutoCAD command has been canceled.
:vlr-commandFailed	An AutoCAD command failed to complete.
:vlr-lispWillStart	An AutoLISP expression is to be evaluated.
:vlr-lispEnded	Evaluation of an AutoLISP expression has completed.
:vlr-lispCancelled	Evaluation of an AutoLISP expression has been canceled.
:vlr-beginDwgOpen	AutoCAD is about to open a drawing file.
:vlr-endDwgOpen	AutoCAD has ended the open operation.
:vlr-dwgFileOpened	A new drawing has been loaded into the AutoCAD drawing window.
:vlr-beginSave	AutoCAD is about to save the drawing file.
:vlr-saveComplete	AutoCAD has saved the current drawing to disk.
:vlr-sysVarWillChange	AutoCAD is about to change the value of a system variable.
:vlr-sysVarChanged	The value of a system variable has changed.

Editor reactor callback data

Name	List length	Parameters
:vlr-lispEnded :vlr-lispCancelled :vlr-beginClose :vlr-beginDxfIn :vlr-abortDxfIn :vlr-dxfInComplete :vlr-beginDxfOut :vlr-abortDxfOut :vlr-dxfOutComplete :vlr-databaseToBeDestroyed	0	
:vlr-unknownCommand :vlr-commandWillStart :vlr-commandEnded :vlr-commandCancelled :vlr-commandFailed	1	A string containing the command name.
:vlr-lispWillStart	1	A string containing the first line of the AutoLISP expression to evaluate.
:vlr-beginDwgOpen :vlr-endDwgOpen :vlr-dwgFileOpened	1	A string identifying the file to open.
:vlr-beginSave	1	A string containing the default file name for save; this may be changed by the user.
:vlr-saveComplete	1	A string identifying the actual file name used for the save.
:vlr-sysVarWillChange	1	A string naming the system variable.
:vlr-sysVarChanged	2	First parameter is a string naming the system variable. Second parameter is an integer indicating whether the change was successful (1 = success, 0 = failed).

vlr-insert-reactor

Constructs an editor reactor object that notifies of an event related to block insertion

```
(vlr-insert-reactor data callbacks)
```

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "Insert reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "Insert reactor callback data."

Return Values

The *reactor_object* argument.

Insert reactor events

Event name	Description
:vlr-beginInsert	A block is about to be inserted into the drawing database.
:vlr-beginInsertM	A 3D transformation matrix is about to be inserted into the drawing database.
:vlr-otherInsert	A block or matrix has been added to the drawing database. This notification is sent after the insert process completes copying the object into the database, but before ID translation or entity transformation occurs.

Insert reactor events (*continued*)

Event name	Description
:vlr-endInsert	Usually indicates an insert operation on the drawing database is complete. However, in some cases, the transform has not yet happened, or the block that was created has not yet been appended. This means the objects copied are not yet graphical, and you cannot use them in selection sets until the :vlr-commandEnded notification is received.
:vlr-abortInsert	Insert operation was terminated and did not complete, leaving the database in an unstable state.

Insert reactor callback data

Name	List length	Parameters
:vlr-beginInsert	3	First parameter is a VLA-object pointing to the database in which the block is being inserted. Second parameter is a string naming the block to be inserted. Third parameter is a VLA-object identifying the source database of the block.
:vlr-beginInsertM	3	First parameter is a VLA-object pointing to the database in which the 3D transformation matrix is being inserted. Second parameter is the 3D transformation matrix to be inserted. Third parameter is a VLA-object identifying the source database of the matrix.
:vlr-otherInsert	2	First parameter is a VLA-object pointing to the database in which the block or matrix is being inserted. Second parameter is a VLA-object identifying the source database of the block or matrix.
:vlr-endInsert :vlr-abortInsert	1	VLA-object pointing to target database.

vlr-linker-reactor

Constructs a reactor object that notifies your application every time an ObjectARX application is loaded or unloaded

```
(vlr-linker-reactor data callbacks)
```

Arguments

data	Any AutoLISP data to be associated with the reactor object.
callbacks	A list of pairs of the following form: (e*vent-name* . *callback_function*) where *event-name* is one of the symbols listed in the table "Linker reactor events" on page 328, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments: *reactor_object* The VLR object that called the callback function *list* A list containing the name of the ObjectARX program that was loaded or unloaded (a string).

Return Values

The *reactor_object* argument.

Linker reactor events

Name	Event
:vlr-rxAppLoaded	The dynamic linker has loaded a new ObjectARX program. The program has finished its initialization.
:vlr-rxAppUnLoaded	The dynamic linker has unloaded an ObjectARX program. The program already has done its clean-up.

Examples

```
_$ (vlr-linker-reactor nil
   '((:vlr-rxAppLoaded . my-vlr-trace-reaction)))
#<VLR-Linker-Reactor>
```

vlr-lisp-reactor

Constructs an editor reactor object that notifies of a LISP event

```
(vlr-lisp-reactor data callbacks)
```

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form: (e*vent-name* . *callback_function*) where *event-name* is one of the symbols listed in the "Lisp reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments: *reactor_object* The VLR object that called the callback function. *list* A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "Lisp reactor callback data."

Return Values

The *reactor_object* argument.

Lisp reactor events

Event name	Description
:vlr-lispWillStart	An AutoLISP expression is to be evaluated.
:vlr-lispEnded	Evaluation of an AutoLISP expression has completed.
:vlr-lispCancelled	Evaluation of an AutoLISP expression has been canceled.

Lisp reactor callback data

Name	List length	Parameters
:vlr-lispEnded :vlr-lispCancelled	0	
:vlr-lispWillStart	1	A string containing the first line of the AutoLISP expression to evaluate.

vlr-miscellaneous-reactor

Constructs an editor reactor object that does not fall under any other editor reactor types

```
(vlr-miscellaneous-reactor data callbacks)
```

Arguments

data Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.

callbacks A list of pairs of the following form:

(*event-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "Miscellaneous reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function.

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "Miscellaneous reactor callback data."

Return Values

The *reactor_object* argument.

Miscellaneous reactor events

Event name	Description
:vlr-pickfirstModified	The pickfirst selection set of the current document has been modified.
:vlr-layoutSwitched	The layout was switched.

Miscellaneous reactor callback data

Name	List length	Parameters
:vlr-pickfirstModified	0	
:vlr-layoutSwitched	1	A string naming the layout switched to.

vlr-mouse-reactor

Constructs an editor reactor object that notifies of a mouse event (for example, a double-click)

(vlr-mouse-reactor *data callbacks*)

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "Mouse reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function *list* A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "Mouse reactor callback data."

Return Values

The *reactor_object* argument.

Mouse reactor events

Event name	Description
:vlr-beginDoubleClick	The user has double-clicked.
:vlr-beginRightClick	The user has right-clicked.

Mouse reactor callback data

Name	List length	Parameters
:vlr-beginDoubleClick :vlr-beginRightClick	1	A 3D point list (list of 3 reals) showing the point clicked on, in WCS.

vlr-notification

Determines whether or not a reactor will fire if its associated namespace is not active

(vlr-notification *reactor*)

Arguments

reactor	A VLR object.

Return Values

A symbol, which can be either `'all-documents` (the reactor fires whether or not its associated document is active), or `'active-document-only` (the reactor fires only if its associated document is active).

vlr-object-reactor

Constructs an object reactor object

```
(vlr-object-reactor owners data callbacks)
```

The reactor object is added to the drawing database, but does not become persistent.

Arguments

owners	An AutoLISP list of VLA-objects identifying the drawing objects to be watched.
data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the table “Object events” on page 334, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts three arguments:

owner The owner of the VLA-object the event applies to.

reactor_object The VLR object that called the callback function.

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table “Object events callback data” on page 335.

Return Values

The *reactor_object* argument.

Object events

Name	Event
:vlr-cancelled	The modification of the object has been canceled.
:vlr-copied	The object has been copied.
:vlr-erased	Erase-flag of the object has been set.
:vlr-unerased	Erase-flag of the object has been reset.
:vlr-goodbye	The object is about to be deleted from memory.
:vlr-openedForModify	The object is about to be modified.
:vlr-modified	The object has been modified. If the modification was canceled, also :vlr-cancelled and :vlr-modifyUndone will be fired.
:vlr-subObjModified	A sub-entity of the object has been modified. This event is triggered for modifications to vertices of polylines or meshes, and for attributes owned by blockReferences.
:vlr-modifyUndone	The object's modification was undone.
:vlr-modifiedXData	The object's extended entity data have been modified.
:vlr-unappended	The object has been detached from the drawing database.
:vlr-reappended	The object has been re-attached to the drawing database.
:vlr-objectClosed	The object's modification has been finished.

Object events callback data

Name	List length	Parameters
:vlr-cancelled :vlr-erased, :vlr-unerased :vlr-goodbye :vlr-openedForModify :vlr-modified :vlr-modifyUndone :vlr-modifiedXData :vlr-unappended :vlr-reappended :vlr-objectClosed	0	
:vlr-copied	1	The object created by the copy operation (ename).
:vlr-subObjModified	1	The sub-object (ename) that has been modified.

Examples

The following code attaches an object reactor to the `myCircle` object. It defines the reactor to respond whenever the object is modified (`:vlr-modified`) and to call the **`print-radius`** function in response to the modification event:

```
(setq circleReactor (vlr-object-reactor (list myCircle)
          "Circle Reactor" '((:vlr-modified . print-radius))))
```

vlr-owner-add

Adds an object to the list of owners of an object reactor

`(vlr-owner-add` ***`reactor owner`*****`)`**

This function adds a new source of reactor events; the reactor will receive events from the specified object.

Arguments

reactor	A VLR object.
owner	A VLA-object to be added to the list of notifiers for this reactor.

Return Values

The VLA-object to which the reactor has been added.

Examples

In the following example, an arc object named "archie" is added to the owner list of reactor `circleReactor`:

```
_$ (vlr-owner-add circleReactor archie)
#<VLA-OBJECT IAcadArc 03ad0bcc>
```

See Also

The vlr-owner-remove function.

vlr-owner-remove

Removes an object from the list of owners of an object reactor

```
(vlr-owner-remove reactor owner)
```

Arguments

reactor	A VLR object.
owner	A VLA-object to be removed from the list of notifiers for this reactor.

Return Values

The VLA-object from which the reactor was removed.

Examples

```
_$ (vlr-owner-remove circleReactor archie)
#<VLA-OBJECT IAcadArc 03ad0bcc>
```

See Also

The vlr-owner-add function.

vlr-owners

Returns the list of owners of an object reactor

(vlr-owners *reactor*)

Arguments

reactor	A VLR object.

Return Values

A list of objects that notify the specified reactor.

Examples

```
 _$ (vlr-owners circleReactor)
(#<VLA-OBJECT IAcadCircle 01db98f4> #<VLA-OBJECT IAcadCircle
01db9724> #<VLA-OBJECT IAcadCircle 01db93d4> #<VLA-OBJECT
IAcadCircle 01db9084>)
```

vlr-pers

Makes a reactor persistent

(vlr-pers *reactor*)

Arguments

reactor	A VLR object.

Return Values

The specified reactor object, if successful, `nil` otherwise.

Examples

Define a reactor:

```
_$ (setq circleReactor (vlr-object-reactor
(list myCircle) "Radius size" '((:vlr-modified . print-radius)))
#<VLR-Object-Reactor>
```

Make the reactor persistent:

```
_$ (vlr-pers circleReactor)
#<VLR-Object-Reactor>
```

vlr-pers-list

Returns a list of persistent reactors in the current drawing document

(vlr-pers-list *[reactor]*)

Arguments

reactor	The reactor object to be listed. If ***reactor*** is not specified, **vlr-pers-list** lists all persistent reactors.

Return Values

A list of reactor objects.

Examples

```
_$ (vlr-pers-list)
(#<VLR-Object-Reactor> #<VLR-Object-Reactor>
(#<VLR-Object-Reactor>)
```

vlr-pers-p

Determines whether or not a reactor is persistent

(vlr-pers-p *reactor*)

Arguments

reactor	A VLR object.

Return Values

The specified reactor object, if it is persistent; `nil`, if the reactor is transient.

Examples

Make a reactor persistent:

```
_$ (vlr-pers circleReactor)
#<VLR-Object-Reactor>
```

Verify that a reactor is persistent:

```
_$ (vlr-pers-p circleReactor)
#<VLR-Object-Reactor>
```

Change the persistent reactor to transient:

```
_$ (vlr-pers-release circleReactor)
#<VLR-Object-Reactor>
```

Verify that the reactor is no longer persistent:

```
_$ (vlr-pers-p circleReactor)
nil
```

vlr-pers-release

Makes a reactor transient

```
(vlr-pers-release reactor)
```

Arguments

reactor	VLR object.

Return Values

The specified reactor object, if `successful`, `nil` otherwise.

vlr-reaction-name

Returns a list of all possible callback conditions for this reactor type

```
(vlr-reaction-names reactor-type)
```

Arguments

reactor-type One of the following symbols:

:VLR-AcDb-Reactor
:VLR-Command-Reactor
:VLR-DeepClone-Reactor
:VLR-DocManager-Reactor
:VLR-DWG-Reactor
:VLR-DXF-Reactor
:VLR-Editor-Reactor
:VLR-Insert-Reactor
:VLR-Linker-Reactor
:VLR-Lisp-Reactor
:VLR-Miscellaneous-Reactor

:VLR-Mouse-Reactor
:VLR-Object-Reactor
:VLR-SysVar-Reactor
:VLR-Toolbar-Reactor
:VLR-Undo-Reactor
:VLR-Wblock-Reactor
:VLR-Window-Reactor
:VLR-XREF-Reactor

Return Values

A list of symbols indicating the possible events for the specified reactor type.

Examples

```
_$ (vlr-reaction-names :VLR-Editor-Reactor)
(:vlr-unknownCommand :vlr-commandWillStart :vlr-commandEnded....
```

vlr-reaction-set

Adds or replaces a callback function in a reactor

(vlr-reaction-set *reactor event function*)

Arguments

reactor	A VLR object.
event	A symbol denoting one of the event types available for this reactor type.
function	A symbol representing the AutoLISP function to be added or replaced.

Return Values

Unspecified.

Examples

The following command changes the `circleReactor` reactor to call the **`print-area`** function when an object is modified:

```
_$ (vlr-reaction-set circleReactor :vlr-modified 'print-area)
PRINT-AREA
```

vlr-reactions

Returns a list of pairs (event-name . callback_function) for the reactor

`(vlr-reactions` *`reactor`*`)`

Arguments

reactor	A VLR object.

Examples

```
_$ (vlr-reactions circleReactor)
((:vlr-modified . PRINT-RADIUS))
```

vlr-reactors

Returns a list of existing reactors

`(vlr-reactors` *`[reactor-type...]`*`)`

Arguments

reactor-type One or more of the following symbols:

:VLR-AcDb-Reactor
:VLR-Command-Reactor
:VLR-DeepClone-Reactor
:VLR-DocManager-Reactor
:VLR-DWG-Reactor
:VLR-DXF-Reactor
:VLR-Editor-Reactor
:VLR-Insert-Reactor
:VLR-Linker-Reactor
:VLR-Lisp-Reactor
:VLR-Miscellaneous-Reactor
:VLR-Mouse-Reactor
:VLR-Object-Reactor
:VLR-SysVar-Reactor
:VLR-Toolbar-Reactor
:VLR-Undo-Reactor
:VLR-Wblock-Reactor
:VLR-Window-Reactor
:VLR-XREF-Reactor

If you specify *reactor-type* arguments, **`vlr-reactors`** returns lists of the reactor types you specified. If you omit *reactor-type*, **`vlr-reactors`** returns all existing reactors.

Return Values

A list of reactor lists, or `nil`, if there are no reactors of any specified type. Each reactor list begins with a symbol identifying the reactor type, followed by pointers to each reactor of that type.

Examples

List all reactors in a drawing:

```
_$ (vlr-reactors)
((:VLR-Object-Reactor #<VLR-Object-Reactor>) (:VLR-Editor-Reactor
#<VLR-Editor-Reactor>))
```

List all object reactors:

```
_$ (vlr-reactors :vlr-object-reactor)
((:VLR-Object-Reactor #<VLR-Object-Reactor>))
```

`vlr-reactors` returns a list containing a single reactor list.

List all database reactors:

```
_$ (vlr-reactors :vlr-acdb-reactor)
nil
```

There are no database reactors defined.

List all DWG reactors:

```
_$ (vlr-reactors :vlr-dwg-reactor)
((:VLR-DWG-Reactor #<VLR-DWG-Reactor> #<VLR-DWG-Reactor>))
```

`vlr-reactors` returns a list containing a list of DWG reactors.

vlr-remove

Disables a reactor

`(vlr-remove` ***`reactor`*****`)`**

Arguments

reactor	A VLR object.

Return Values

The *reactor* argument, or `nil`, if unsuccessful.

Examples

The following command disables the `circleReactor` reactor:

```
_$ (vlr-remove circleReactor)
#<VLR-Object-reactor>
```

See Also

The vlr-remove-all function.

vlr-remove-all

Disables all reactors of the specified type

```
(vlr-remove-all [reactor-type])
```

Arguments

reactor-type	One of the following symbols: :VLR-AcDb-Reactor :VLR-Command-Reactor :VLR-DeepClone-Reactor :VLR-DocManager-Reactor :VLR-DWG-Reactor :VLR-DXF-Reactor :VLR-Editor-Reactor :VLR-Insert-Reactor :VLR-Linker-Reactor :VLR-Lisp-Reactor :VLR-Miscellaneous-Reactor :VLR-Mouse-Reactor :VLR-Object-Reactor :VLR-SysVar-Reactor :VLR-Toolbar-Reactor :VLR-Undo-Reactor :VLR-Wblock-Reactor :VLR-Window-Reactor :VLR-XREF-Reactor If no *reactor-type* is specified, **`vlr-remove-all`** disables all reactors.

Return Values

A list of lists. The first element of each list identifies the type of reactor, and the remaining elements identify the disabled reactor objects. The function returns `nil` if there are no reactors active.

Examples

The following function call disables all editor reactors:

```
_$ (vlr-remove-all :vlr-editor-reactor)
((:VLR-Editor-Reactor #<VLR-Editor-Reactor>))
```

The following call disables all reactors:

```
_$ (vlr-remove-all)
((:VLR-Object-Reactor #<VLR-Object-Reactor> #<VLR-Object-Reactor>
#<VLR-Object-Reactor>) (:VLR-Editor-Reactor #<VLR-Editor-Reactor>))
```

See Also

The vlr-remove function.

vlr-set-notification

Defines whether or not a reactor's callback function will execute if its associated namespace is not active

```
(vlr-set-notification reactor 'range)
```

Arguments

reactor	A VLR object.
'range	The *range* argument is a symbol that can be either *'all-documents* (execute the callback whether or not the reactor is associated with the active document), or *'active-document-only* (execute the callback only if the reactor is associated with the active document).

Return Values

The VLR object.

Examples

Set a reactor to execute its callback function even if its associated namespace is not active:

```
_$ (vlr-set-notification circleReactor 'all-documents)
```

```
#<VLR-Object-Reactor>
```

vlr-sysvar-reactor

Constructs an editor reactor object that notifies of a change to a system variable

`(vlr-sysvar-reactor` *`data callbacks`*`)`

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form: (e*vent-name* . *callback_function*) where *event-name* is one of the symbols listed in the "SysVar reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments: *reactor_object* The VLR object that called the callback function. *list* A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "SysVar reactor callback data."

Return Values

The *reactor_object* argument.

SysVar reactor events

Event name	Description
:vlr-sysVarWillChange	AutoCAD is about to change the value of a system variable.
:vlr-sysVarChanged	The value of a system variable has changed.

SysVar reactor callback data

Name	List length	Parameters
:vlr-sysVarWillChange	1	A string identifying the system variable name.
:vlr-sysVarChanged	2	First parameter is a string identifying the system variable name. Second parameter is symbol indicating whether or not the change was successful (`T` if successful, `nil` if not).

vlr-toolbar-reactor

Constructs an editor reactor object that notifies of a change to the bitmaps in a toolbar

(vlr-toolbar-reactor *data callbacks*)

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the “Toolbar reactor events” table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function.

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table “Toolbar reactor callback data.”

Return Values

The *reactor_object* argument.

Toolbar reactor events

Event name	Description
:vlr-toolbarBitmapSizeWillChange	The size of the AutoCAD toolbar icons is about to change.
:vlr-toolbarBitmapSizeChanged	The size of the AutoCAD toolbar icons has changed.

Toolbar reactor callback data

Name	List length	Parameters
:vlr-toolbarBitmapSizeWillChange :vlr-toolbarBitmapSizeChanged	1	`T`, if the toolbar is being set to large bitmaps, `nil` if the toolbar is being set to small bitmaps.

vlr-trace-reaction

A pre-defined callback function that prints one or more callback arguments in the Trace window

```
(vlr-trace-reaction)
```

This function can be used as a debugging tool to verify that a reactor has fired.

Examples

Define a command reactor and assign **vlr-trace-reaction** as the callback function:

```
_$  (VLR-Reaction-Set (VLR-Command-Reactor) :VLR-commandWillStart
'VLR-trace-reaction)
VLR-trace-reaction
```

At the AutoCAD Command prompt, enter the following:

```
 _.LINE
```

Respond to the command prompts, then activate the VLISP window and open the Trace window. You should see the following in the Trace window:

```
; "Reaction": :VLR-commandWillStart; "argument list":
(#<VLR-COMMAND-REACTOR> ("LINE"))
```

The output from **`vlr-trace-reaction`** identifies the type of trigger event, the reactor type, and the command that triggered the reactor.

vlr-type

Returns a symbol representing the reactor type

`(vlr-type` ***`reactor`*****`)`**

Arguments

reactor	A VLR object.

Return Values

A symbol identifying the reactor type. The following table lists the types that may be returned by **`vlr-type`**:

Reactor types

Reactor type	Description
:VLR-AcDb-Reactor	Database reactor.
:VLR-Command-Reactor	An editor reactor notifying of a command event.
:VLR-DeepClone-Reactor	An editor reactor notifying of a deep clone event.
:VLR-DocManager-Reactor	Document management reactor.
:VLR-DWG-Reactor	An editor reactor notifying of a drawing event (for example, opening or closing a drawing file).
:VLR-DXF-Reactor	An editor reactor notifying of an event related to reading or writing of a DXF file.
:VLR-Editor-Reactor	General editor reactor; maintained for backward-compatibility.
:VLR-Insert-Reactor	An editor reactor notifying of an event related to block insertion.

Reactor types (*continued*)

Reactor type	Description
:VLR-Linker-Reactor	Linker reactor.
:VLR-Lisp-Reactor	An editor reactor notifying of a LISP event.
:VLR-Miscellaneous-Reactor	An editor reactor that does not fall under any of the other editor reactor types.
:VLR-Mouse-Reactor	An editor reactor notifying of a mouse event (for example, a double-click).
:VLR-Object-Reactor	Object reactor.
:VLR-SysVar-Reactor	An editor reactor notifying of a change to a system variable.
:VLR-Toolbar-Reactor	An editor reactor notifying of a change to the bitmaps in a toolbar.
:VLR-Undo-Reactor	An editor reactor notifying of an undo event.
:VLR-Wblock-Reactor	An editor reactor notifying of an event related to writing a block.
:VLR-Window-Reactor	An editor reactor notifying of an event related to moving or sizing an AutoCAD window.
:VLR-XREF-Reactor	An editor reactor notifying of an event related to attaching or modifying XREFs.

Examples

```
_$ (vlr-type circleReactor)
:VLR-Object-Reactor
```

vlr-types

Returns a list of all reactor types

```
(vlr-types)
```

Return Values

```
(:VLR-Linker-Reactor :VLR-Editor-Reactor :VLR-AcDb-Reactor ....)
```

vlr-undo-reactor

Constructs an editor reactor object that notifies of an undo event

```
(vlr-undo-reactor data callbacks)
```

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form:

(*event-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "Undo reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function.

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "Undo reactor callback data."

Return Values

The *reactor_object* argument.

Undo reactor events

Event name	Description
:vlr-undoSubcommandAuto	The UNDO command's Auto option has been executed.
:vlr-undoSubcommandControl	The UNDO command's Control option has been executed.
:vlr-undoSubcommandBegin	The UNDO command's BEGIN or GROUP option is being performed. BEGIN and GROUP mark the beginning of a series of commands that can be undone as one unit.
:vlr-undoSubcommandEnd	The UNDO command's END option is being performed. UNDO/END marks the end of a series of commands that can be undone as one unit.

Undo reactor events (*continued*)

Event name	Description
:vlr-undoSubcommandMark	The UNDO command's MARK option is about to be executed. This places a marker in the undo file so UNDO/BACK can undo back to the marker.
:vlr-undoSubcommandBack	The UNDO command's BACK option is about to be performed. UNDO/BACK undoes everything back to the most recent MARK marker or back to the beginning of the undo file if no MARK marker exists.
:vlr-undoSubcommandNumber	The UNDO command's NUMBER option is about to be executed (the default action of the UNDO command).

Undo reactor callback data

Name	List length	Parameters
:vlr-undoSubcommandAuto	2	First parameter is an integer indicating the activity. The value is always 4, indicating that notification occurred after the operation was performed. Second parameter is a symbol indicating the state of Auto mode. Value is `T` if Auto mode is turned on, `nil` if Auto mode is turned off.
:vlr-undoSubcommandControl	2	First parameter is an integer indicating the activity. The value is always 4, indicating that notification occurred after the operation was performed. Second parameter is an integer indicating the Control option selected. This can be one of the following: 0—NONE was selected 1—ONE was selected 2—ALL was selected
:vlr-undoSubcommandBegin :vlr-undoSubcommandEnd :vlr-undoSubcommandMark :vlr-undoSubcommandBack	1	An integer value of 0, indicating that notification occurs before the actual operation is performed.
:vlr-undoSubcommandNumber	2	First parameter is an integer indicating the activity. The value is always 0, indicating that notification occurs before the actual operation is performed. Second parameter is an integer indicating the number of steps being undone.

vlr-wblock-reactor

Constructs an editor reactor object that notifies of an event related to writing a block

```
(vlr-wblock-reactor data callbacks)
```

Arguments

data Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.

callbacks A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "Wblock reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function.

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "Wblock reactor callback data."

Return Values

The *reactor_object* argument.

Wblock reactor events

Event name	Description
:VLR-wblockNotice	A wblock operation is about to start.
:VLR-beginWblockPt	A wblock operation is being performed on a set of entities.
:VLR-beginWblockId	A wblock operation is being performed on a specified block.

Wblock reactor events (*continued*)

Event name	Description
:VLR-beginWblock	A wblock operation is being performed on an entire database. Notification does not occur until all the entities in the source database's model space are copied into the target database.
:VLR-otherWblock	A wblock operation is being performed on a drawing database. This notification is sent after the wblock process completes copying the objects into the target database, but before ID translation occurs. At this time it is possible to clone additional objects (such as dictionaries and objects that reside in dictionaries that would otherwise not be copied over) in the same way as during beginDeepCloneXlation notification.
:VLR-abortWblock	A wblock operation was terminated before completing.
:VLR-endWblock	A wblock operation completed successfully.
:VLR-beginWblockObjects	wblock has just initialized the object ID translation map.

Wblock reactor callback data

Name	List length	Parameters
:VLR-wblockNotice	1	Database object (VLA-object) from which the block will be created.
:VLR-beginWblockPt	3	First parameter is the target database object (VLA-object). Second parameter is the source database object (VLA-object) containing the objects being wblocked. Third parameters is a 3D point list (in WCS) to be used as the base point in the target database.
:VLR-beginWblockId	3	First parameter is the target database object (VLA-object). Second parameter is the source database object (VLA-object) containing the objects being wblocked. Third parameter is the object ID of the BlockTableRecord being wblocked.

Wblock reactor callback data (*continued*)

Name	List length	Parameters
:VLR-beginWblock :VLR-otherWblock	2	First parameter is the target database object (VLA-object). Second parameter is the source database object (VLA-object) containing the objects being wblocked.
:VLR-abortWblock :VLR-endWblock	1	The target database object (VLA-object).
:VLR-beginWblockObjects	2	First parameter is the source database object (VLA-object) containing the objects being wblocked. Second parameter is an ID map.

vlr-window-reactor

Constructs an editor reactor object that notifies of an event related to moving or sizing an AutoCAD window

(vlr-window-reactor *data callbacks*)

Arguments

data Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.

callbacks A list of pairs of the following form:

(e*vent-name* . *callback_function*)

where *event-name* is one of the symbols listed in the "Window reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function.

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "Window reactor callback data."

Return Values

The *reactor_object* argument.

Window reactor events

Event name	Description
:vlr-docFrameMovedOrResized	An MDI child frame window (a document window) has been moved or resized.
:vlr-mainFrameMovedOrResized	The main AutoCAD window has been moved or resized.

Window reactor callback data

Name	List length	Parameters
:vlr-docFrameMovedOrResized :vlr-mainFrameMovedOrResized	2	The first parameter is an integer containing the HWND of the window. The second parameter indicates whether the window has been moved or resized. The value is `T` if the window has been moved, `nil` if the window has been resized.

vlr-xref-reactor

Constructs an editor reactor object that notifies of an event related to attaching or modifying XREFs

(vlr-xref-reactor *data callbacks*)

Arguments

data	Any AutoLISP data to be associated with the reactor object, or `nil`, if no data.
callbacks	A list of pairs of the following form: (*event-name* . *callback_function*) where *event-name* is one of the symbols listed in the "XREF reactor events" table below, and *callback_function* is a symbol representing a function to be called when the event fires. Each callback function accepts two arguments:

reactor_object The VLR object that called the callback function.

list A list of extra data elements associated with the particular event. The contents of this list for particular events is shown in the table "XREF reactor callback data."

Return Values

The *reactor_object* argument.

XREF reactor events

Event name	Description
:VLR-beginAttach	An XREF is about to be attached.
:VLR-otherAttach	An external reference is being added to the drawing database. This event occurs after objects are cloned, but before any translation. This callback function is sent just after beginDeepCloneXlation notification, but only occurs for the XREF attach process.
:VLR-abortAttach	An XREF attach operation was terminated before successful completion.
:VLR-endAttach	An XREF attach operation completed successfully.
:VLR-redirected	An object ID in the XREF drawing is being modified to point to the associated object in the drawing being XREFed into.
:VLR-comandeered	The object ID of the object is being appended to the symbol table of the drawing being XREFed into.
:VLR-beginRestore	An existing XREF is about to be resolved (typically when a drawing with XREFs is loading).
:VLR-abortRestore	An XREF unload or reload was terminated before successful completion.
:VLR-endRestore	An existing XREF has been resolved (typically when a drawing with XREFs has completed loading).
:VLR-xrefSubcommandBindItem	The BIND subcommand of XREF was invoked, or a pre-existing xref is being bound. Note that the BIND subcommand is interactive and triggers multiple events.

XREF reactor events (*continued*)

Event name	Description
:VLR-xrefSubcommandAttachItem	The ATTACH subcommand of XREF was invoked, or a pre-existing xref is being resolved. Note that the ATTACH subcommand is interactive and triggers multiple events.
:VLR-xrefSubcommandOverlayItem	The OVERLAY subcommand of XREF was invoked, or a pre-existing xref is being resolved. Note that the OVERLAY subcommand is interactive and triggers multiple events.
:VLR-xrefSubcommandDetachItem	The DETACH subcommand of XREF was invoked. Note that the OVERLAY subcommand is interactive and triggers multiple events.
:VLR-xrefSubcommandPathItem	The PATH subcommand of XREF was invoked. Note that the PATH subcommand is interactive and triggers multiple events.
:VLR-xrefSubcommandReloadItem	The RELOAD subcommand of XREF was invoked, or a pre-existing xref is being reloaded. Note that the RELOAD subcommand is interactive and triggers multiple events.
:VLR-xrefSubcommandUnloadItem	The UNLOAD subcommand of XREF was invoked, or a pre-existing xref is being unloaded.

XREF reactor callback data

Name	List length	Parameters
:VLR-beginAttach	3	First parameter is a VLA-object pointing to the target drawing database. Second parameter is a string containing the file name of the xref being attached. Third parameter is a VLA-object pointing to the drawing database that contains the objects being attached.
:VLR-otherAttach	2	First parameter is a VLA-object pointing to the target drawing database. Second parameter is a VLA-object pointing to the drawing database that contains the objects being attached.

XREF reactor callback data (*continued*)

Name	List length	Parameters
:VLR-abortAttach	1	A VLA-object pointing to the drawing database that contains the objects being attached.
:VLR-endAttach	1	A VLA-object pointing to the target drawing database.
:VLR-redirected	2	First parameter is an integer containing the object ID for the redirected symbol table record (STR) in the drawing being XREFed to. Second parameter is an integer containing the object ID for the object in the xref drawing.
:VLR-comandeered	3	First parameter is a VLA-object pointing to the database receiving the XREF. Second parameter is an integer containing the object ID of the object being commandeered. Third parameter is a VLA-object pointing to the drawing database that contains the objects being attached.
:VLR-beginRestore	3	First parameter is a VLA-object pointing to the database receiving the XREF. Second parameter is a string containing the XREF block table record (BTR) name. Third parameter is a VLA-object pointing to the drawing database that contains the objects being attached.
:VLR-abortRestore :VLR-endRestore	1	A VLA-object pointing to the target drawing database.

XREF reactor callback data (*continued*)

Name	List length	Parameters
:VLR-xrefSubcommandBindItem	2	First parameter is an integer indicating the activity the BIND is carrying out. Possible values are: 0—BIND subcommand invoked. 2—xref with the indicated object ID is being bound. 3—xref with the indicated object ID was successfully bound. 4—BIND subcommand completed. 5—BIND operation is about to either terminate or fail to complete on the specified object ID. 6—BIND operation has either terminated or failed to complete on the specified object ID. 7—Sent for an XDep block bound by XBind. 8—Sent for all other symbols: Layers, Linetypes, TextStyles, and DimStyles. Second parameter is an integer containing the object ID of the xref being bound, or 0 if not applicable.
:VLR-xrefSubcommandAttachItem	2	First parameter is an integer indicating the activity the ATTACH is carrying out. Possible values are: 0—BIND subcommand invoked. 2—xref with the indicated object ID is being bound. 3—xref with the indicated object ID was successfully bound. 4—BIND subcommand completed. 5—BIND operation is about to either terminate or fail to complete on the specified object ID. 6—BIND operation has either terminated or failed to complete on the specified object ID. Second parameter is a string identifying the file being attached, or `nil` if not applicable.

XREF reactor callback data (*continued*)

Name	List length	Parameters
:VLR-xrefSubcommandOverlayItem	2	First parameter is an integer indicating the activity the OVERLAY is carrying out. Possible values are: 0—BIND subcommand invoked. 2—xref with the indicated object ID is being bound. 3—xref with the indicated object ID was successfully bound. 4—BIND subcommand completed. 5—BIND operation is about to either terminate or fail to complete on the specified object ID. 6—BIND operation has either terminated or failed to complete on the specified object ID. Second parameter is a string identifying the file being overlaid, or `nil` if not applicable.
:VLR-xrefSubcommandDetachItem	2	First parameter is an integer indicating the activity the DETACH is carrying out. Possible values are: 0—BIND subcommand invoked. 2—xref with the indicated object ID is being bound. 3—xref with the indicated object ID was successfully bound. 4—BIND subcommand completed. 5—BIND operation is about to either terminate or fail to complete on the specified object ID. 6—BIND operation has either terminated or failed to complete on the specified object ID. Second parameter is an integer containing the object ID of the xref being detached, or 0 if not applicable.

XREF reactor callback data (*continued*)

Name	List length	Parameters
:VLR-xrefSubcommandPathItem	3	First parameter is an integer indicating the activity the DETACH is carrying out. Possible values are: 0—BIND subcommand invoked. 2—xref with the indicated object ID is being bound. 3—xref with the indicated object ID was successfully bound. 4—BIND subcommand completed. 5—BIND operation is about to either terminate or fail to complete on the specified object ID. 6—BIND operation has either terminated or failed to complete on the specified object ID. Second parameter is an integer containing the object ID of the xref being operated on, or 0 if not applicable. Third parameter is a string identifying the new path name of the xref, or `nil` if not applicable.
:VLR-xrefSubcommandReloadItem	2	First parameter is an integer indicating the activity the RELOAD is carrying out. Possible values are: 0—BIND subcommand invoked. 2—xref with the indicated object ID is being bound. 3—xref with the indicated object ID was successfully bound. 4—BIND subcommand completed. 5—BIND operation is about to either terminate or fail to complete on the specified object ID. 6—BIND operation has either terminated or failed to complete on the specified object ID. Second parameter is an integer containing the object ID of the xref being reloaded, or 0 if not applicable.

XREF reactor callback data (*continued*)

Name	List length	Parameters
:VLR-xrefSubcommandUnloadItem	2	First parameter is an integer indicating the activity the UNLOAD is carrying out. Possible values are: 0—BIND subcommand invoked. 2—xref with the indicated object ID is being bound. 3—xref with the indicated object ID was successfully bound. 4—BIND subcommand completed. 5—BIND operation is about to either terminate or fail to complete on the specified object ID. 6—BIND operation has either terminated or failed to complete on the specified object ID. Second parameter is an integer containing the object ID of the xref being unloaded, or 0 if not applicable.

vports

Returns a list of viewport descriptors for the current viewport configuration

```
(vports)
```

Return Values

One or more viewport descriptor lists consisting of the viewport identification number and the coordinates of the viewport's lower-left and upper-right corners.

If the AutoCAD TILEMODE system variable is set to 1 (on), the returned list describes the viewport configuration created with the AutoCAD VPORTS command. The corners of the viewports are expressed in values between 0.0 and 1.0, with (0.0, 0.0) representing the lower-left corner of the display screen's graphics area, and (1.0, 1.0) the upper-right corner. If TILEMODE is 0 (off), the returned list describes the viewport objects created with the MVIEW command. The viewport object corners are expressed in paper space coordinates. Viewport number 1 is always paper space when TILEMODE is off.

Examples

Given a single-viewport configuration with TILEMODE on, the **`vports`** function might return the following:

```
((1 (0.0 0.0) (1.0 1.0)))
```

Given four equal-sized viewports located in the four corners of the screen when TILEMODE is on, the **`vports`** function might return the following lists:

```
((5 (0.5 0.0) (1.0 0.5))
 (2 (0.5 0.5) (1.0 1.0))
 (3 (0.0 0.5) (0.5 1.0))
 (4 (0.0 0.0) (0.5 0.5)) )
```

The current viewport's descriptor is always first in the list. In the previous example, viewport number 5 is the current viewport.

wcmatch

Performs a wild-card pattern match on a string

`(wcmatch` ***`string pattern`*****`)`**

Arguments

string	A string to be compared. The comparison is case-sensitive, so upper- and lowercase characters must match.
pattern	A string containing the pattern to match against *string*. The *pattern* can contain the wild-card pattern-matching characters shown in the table "Wild-card characters" on page 364. You can use commas in a pattern to enter more than one pattern condition. Only the first 500 characters (approximately) of the *string* and *pattern* are compared; anything beyond that is ignored.

Both arguments can be either a quoted string or a string variable. It is valid to use variables and values returned from AutoLISP functions for *string* and *pattern* values.

Return Values

If *string* and *pattern* match, **`wcmatch`** returns `T`, otherwise, **`wcmatch`** returns `nil`.

Wild-card characters

Character	Definition
`#` (pound)	Matches any single numeric digit
`@` (at)	Matches any single alphabetic character
`.` (period)	Matches any single nonalphanumeric character
`*` (asterisk)	Matches any character sequence, including an empty one, and it can be used anywhere in the search pattern: at the beginning, middle, or end
`?` (question mark)	Matches any single character
`~` (tilde)	If it is the first character in the pattern, it matches anything except the pattern
`[...]`	Matches any one of the characters enclosed
`[~...]`	Matches any single character not enclosed
– (hyphen)	Used inside brackets to specify a range for a single character
`,` (comma)	Separates two patterns
`` ` `` (reverse quote)	Escapes special characters (reads next character literally)

Examples

The following command tests a string to see if it begins with the character `N`:

```
Command: (wcmatch "Name" "N*")
T
```

The following example performs three comparisons. If any of the three pattern conditions is met, **`wcmatch`** returns `T`. In this case the tests are: does the string contain three characters; does the string not contain an `m`; and does the string begin with the letter "`N`." If any of the three pattern conditions is met, **`wcmatch`** returns `T`:

```
Command: (wcmatch "Name" "???,~*m*,N*")
T
```

In this example, the last condition was met, so **`wcmatch`** returned `T`.

Using Escape Characters with wcmatch

To test for a wild-card character in a string, you can use the single reverse-quote character (`` ` ``) to *escape* the character. *Escape* means that the character following the single reverse quote is not read as a wild-card character; it is compared at its face value. For example, to search for a comma anywhere in the string "`Name`", enter the following:

```
Command: (wcmatch "Name" "*`,*")
nil
```

Both the C and AutoLISP programming languages use the backslash (\) as an escape character, so you need two backslashes (\\) to produce one backslash in a string. To test for a backslash character anywhere in "`Name`", use the following function call:

```
Command: (wcmatch "Name" "*`\\*")
nil
```

All characters enclosed in brackets (`[ . . . ]`) are read literally, so there is no need to escape them, with the following exceptions: the tilde character (~) is read literally only when it is not the first bracketed character (as in `"[A~BC]"`); otherwise it is read as the negation character, meaning that **`wcmatch`** should match all characters except those following the tilde (as in `"[~ABC]"`). The dash character (–) is read literally only when it is the first or last bracketed character (as in `"[–ABC]"` or `"[ABC–]"`) or when it follows a leading tilde (as in `"[~–ABC]"`). Otherwise, the dash character (–) is used within brackets to specify a range of values for a specific character. The range works only for single characters, so `"STR[1–38]"` matches `STR1`, `STR2`, `STR3`, and `STR8`, and `"[A–Z]"` matches any single uppercase letter.

The closing bracket character (`]`) is also read literally if it is the first bracketed character or if it follows a leading tilde (as in `"[ ]ABC]"` or `"[~]ABC]"`).

NOTE Because additional wild-card characters might be added in future releases of AutoLISP, it is a good idea to escape all nonalphanumeric characters in your pattern to ensure upward compatibility.

while

Evaluates a test expression, and if it is not `nil`, evaluates other expressions; repeats this process until the test expression evaluates to `nil`

`(while` ***`testexpr [expr...]`*** **`)`**

The **`while`** function continues until *testexpr* is `nil`.

Arguments

testexpr	The expression containing the test condition.
expr	One or more expressions to be evaluated until *testexpr* is `nil`.

Return Values

The most recent value of the last *expr*.

Examples

The following code calls user function **`some-func`** ten times, with `test` set to 1 through 10. It then returns 11, which is the value of the last expression evaluated:

```
(setq test 1)
(while (<= test 10)
  (some-func test)
  (setq test (1+ test))
)
```

write-char

Writes one character to the screen or to an open file

`(write-char` ***`num [file-desc]`*** **`)`**

Arguments

num	The decimal ASCII code for the character to be written.
file-desc	A file descriptor for an open file.

Return Values

The *num* argument.

Examples

The following command writes the letter *C* to the command window, and returns the supplied *num* argument:

```
Command: (write-char 67)
C67
```

Assuming that `f` is the descriptor for an open file, the following command writes the letter *C* to that file:

```
Command: (write-char 67 f)
67
```

Note that **`write-char`** cannot write a NULL character (ASCII code 0) to a file.

See Also

The *Customization Guide* for a list of ASCII codes.

write-line

Writes a string to the screen or to an open file

`(write-line` ***`string [file-desc]`*** **`)`**

Arguments

string	A string.
file-desc	A file descriptor for an open file.

Return Values

The *string*, quoted in the normal manner. The quotes are omitted when writing to a file.

Examples

Open a new file:

```
Command: (setq f (open "c:\\my documents\\new.tst" "w"))
#<file "c:\\my documents\\new.tst">
```

Use **`write-line`** to write a line to the file:

```
Command: (write-line "To boldly go where nomad has gone before." f)
"To boldly go where nomad has gone before."
```

The line is not physically written until you close the file:

```
Command: (close f)
nil
```

xdroom

Returns the amount of extended data (xdata) space that is available for an object (entity)

`(xdroom` ***`ename`*****`)`**

Because there is a limit (currently, 16 kilobytes) on the amount of extended data that can be assigned to an entity definition, and because multiple applications can append extended data to the same entity, this function is provided so an application can verify there is room for the extended data that it will append. It can be called in conjunction with xdsize, which returns the size of an extended data list.

Arguments

ename	An entity name (ename data type).

Return Values

An integer reflecting the number of bytes of available space. If unsuccessful, **`xdroom`** returns `nil`.

Examples

The following example that looks up the available space for extended data of a viewport object:

```
Command: (xdroom vpname)
16162
```

In this example, 16,162 bytes of the original 16,383 bytes of extended data space are available, meaning that 221 bytes are used.

xdsize

Returns the size (in bytes) that a list occupies when it is linked to an object (entity) as extended data

(xdsize *lst*)

Arguments

lst	A valid list of extended data that contain an application name previously registered with the use of the **regapp** function. See the "Examples" section of this function for *lst* examples.

Return Values

An integer reflecting the size, in bytes. If unsuccessful, **xdsize** returns `nil`.

Brace fields (group code 1002) must be balanced. An invalid *lst* generates an error and places the appropriate error code in the ERRNO variable. If the extended data contains an unregistered application name, you see this error message (assuming that CMDECHO is on):

Invalid application name in 1001 group

Examples

The *lst* can start with a –3 group code (the extended data sentinel), but it is not required. Because extended data can contain information from multiple applications, the list must have a set of enclosing parentheses.

```
(-3 ("MYAPP" (1000 . "SUITOFARMOR")
             (1002 . "{")
             (1040 . 0.0)
             (1040 . 1.0)
             (1002 . "}")
   )
)
```

Here is the same example without the –3 group code. This list is just the **cdr** of the first example, but it is important that the enclosing parentheses are included:

```
( ("MYAPP" (1000 . "SUITOFARMOR")
           (1002 . "{")
           (1040 . 0.0)
           (1040 . 1.0)
           (1002 . "}")
   )
)
```

zerop

Verifies that a number evaluates to zero

```
(zerop number)
```

Arguments

number	A number.

Return Values

`T` if *number* evaluates to zero, otherwise `nil`.

Examples

Command: **(zerop 0)**
T

Command: **(zerop 0.0)**
T

Command: **(zerop 0.0001)**
nil

Externally Defined Commands

A

AutoCAD® commands defined by ObjectARX® or AutoLISP® applications are called externally defined. AutoLISP applications may need to access externally defined commands differently from the way they access built-in AutoLISP functions. Many externally defined commands have their own programming interfaces that allow AutoLISP applications to take advantage of their functionality.

For additional information on the commands described in this appendix, see the *Command Reference*.

In this chapter

- Alphabetical listing of AutoCAD commands defined by AutoLISP or ObjectARX applications

3dsin

Imports a 3D Studio (*.3ds*) file (Externally-defined: *render* ARX application)

(c:3dsin *mode [multimat create] file*)

Arguments

mode	An integer that specifies whether the command is to be used interactively (`mode` = 1) or noninteractively (`mode` = 0).
multimat	An integer that specifies how to treat objects with multiple materials. Required if *mode* is set to 0. Allowable values are: **0** Create a new object for each material **1** Assign the first material to the new object
create	An integer that specifies how to organize new objects. This mode always imports all the objects in the *.3ds* file. Required if *mode* is set to 0. Allowable values are: **0** Create a layer for each 3DS object **1** Create a layer for each 3DS color **2** Create a layer for each 3DS material **3** Place all new objects on a single layer
file	A string specifying the *.3ds* file to import; the *.3ds* file extension is required.

Mode 0 always imports all the objects in the *.3ds* file.

Examples

Open the 3D Studio file *globe.3ds* for import and prompt the user for import specifics:

```
(c:3dsin 1 "globe.3ds")
```

Import all of *shadow.3ds* with no user input, splitting objects with multiple materials and putting all new objects on the same layer:

```
Command: (c:3dsin 0 0 3 "c:/my documents/cad drawings/shadow.3ds")
Initializing Render...
Initializing preferences...done.
Processing object B_Leg01
Converting material SKIN
Processing object B_Leg02
Processing object Central_01
Processing object Central_02
Processing object F_Leg01
Processing object F_Leg02
Processing object M_Quad01
Processing object ML_Feele01
Processing object ML_Feele02
Processing object Pre_Quad01
Processing object Pre_Quad02
3D Studio file import completed
1
```

3dsout

Exports a 3D Studio file (Externally-defined: *render* ARX application)

(c:3dsout *sset omode div smooth weld file*)

Arguments

sset	A selection set containing the AutoCAD objects to export.
omode	An integer (0 or 1) that specifies the output mode for the representation of AutoCAD data. Currently, **3dsout** output is the same whether *omode* is set to 0 or 1
div	An integer that specifies how to divide AutoCAD objects into 3D Studio objects. Allowable values are: **0** Create one object for each AutoCAD layer **1** Create one object for each AutoCAD color **2** Create one object for each AutoCAD object type
smooth	An integer that specifies the threshold angle for automatic smoothing. If *smooth* is set to –1, no auto-smoothing is done; if set to 0–360, AutoCAD generates smoothing when the angle between face normals is less than this value.

weld	A real number that specifies the distance threshold for welding nearby vertices. If *`weld`* is set to a value less than 0, welding is disabled; if set to a value greater than or equal to 0, AutoCAD welds vertices closer than this value.
file	A string specifying the name of the 3D Studio file to create; the *.3ds* file extension is required.

Examples

Export all of a drawing, creating 3D Studio objects based on drawing layer, using a smoothing threshold of 30 degrees and a welding distance of 0.1:

```
(c:3dsout (ssget "X") 0 0 30 0.1 "testav.3ds")
```

align

Translates and rotates objects, allowing them to be aligned with other objects (Externally-defined: geom3d ARX application)

`(align` ***`arg1 arg2`*** **`...)`**

Arguments

arg1 arg2...	Arguments to the AutoCAD align command. The order, number, and type of arguments for the **`align`** function are the same as if you were entering ALIGN at the command line. To indicate a null response (a user pressing ENTER), specify `nil` or an empty string (`""`).

Return Values

`T` if successful, otherwise `nil`.

Examples

The following example specifies two pairs of source and destination points, which perform a 2D move:

```
(setq ss (ssget))
(align ss s1 d1 s2 d2 "" "2d")
```

cal

Invokes the on-line geometry calculator and returns the value of the evaluated expression (Externally-defined: geomcal ARX application)

(c:cal *expression*)

Arguments

expression	A quoted string. Refer to CAL in the *Command Reference* for a description of allowable expressions.

Return Values

The result of the expression.

Examples

The following example uses **`cal`** in an AutoLISP expression with the **`trans`** function:

```
(trans (c:cal "[1,2,3]+MID") 1 2)
```

fog

Adds distance from the view (Externally-defined: *render* ARX application)

(c:fog *enabled [color [near_dist [far_dist [near_percent [far_percent [background]]]]]]*)

Arguments

enabled	A string that turns fog on and off without affecting other settings. Default is ON.
color	A 3Dpoint specifying a standard AutoCAD color. Default is (111).
near_dist	A real number defining where the fog starts. Default is 0.0.
far_dist	A real number defining where the fog ends. Default is 1.0.
near_percent	A real number defining the percentage of fog at the start of the bank. Default is 0.0.

far_percent	A real number defining the percentage of fog at the end of the bank. Default is 1.0.
background	A string that applies fog to the background as well as to the geometry. Default is OFF (do not apply fog to the background).

With the FOG command, you can provide visual information about the distance of objects from the view's eye. To maximize fog, add white to an image; to maximize depth cueing, add black.

`Nil` or missing trailing arguments are not changed.

light

Creates, modifies, and deletes lights and lighting effects (Externally-defined: *render* ARX application)

`(c:light` ***`mode [options]`*****`)`**

Arguments

mode A string indicating the action to be performed. Allowable *mode* values are:

A	Set or retrieve ambient light intensity
D	Delete existing lights
L	List all lights in the drawing or return a definition of a specified light
M	Modify existing lights
ND	Create a new distant light
NP	Create a new point light
NS	Create a new spotlight
R	Rename an existing light

options The *options* allowed depend on the *mode* and are listed separately for each mode.

NOTE This command is not allowed in paper space.

A—Ambient Light

Set or retrieve the ambient light intensity.

(c:light "A" ***[intensity [color]]*****)**

Arguments

intensity	A real number from 0.0 to 1.0; if *intensity* is omitted, it defaults to 1.0.
color	A list that specifies any RGB triplet; if omitted, it defaults to (1.0 1.0 1.0).

Examples

To set ambient light intensity to 0.6, issue the following:

```
Command: (c:light "A" 0.6)
1
```

To retrieve the current ambient light intensity, omit the *intensity* argument:

```
Command: (c:light "A")
(0.6 (1.0 1.0 1.0))
```

The intensity returned is 0.6, and the color is 1.0 1.0 1.0.

D—Delete Lights

Delete existing lights.

(c:light "D" ***name*****)**

Arguments

name	A string specifying the name of the light to delete.

Examples

The following function call deletes a light named “OLDLGT”:

```
(c:light "D" "OLDLGT")
```

L—List Lights

List all lights in the drawing or return a definition of the specified light.

(c:light "L" ***[name]*****)**

Arguments

name	A string specifying the name of the light to list. If you omit the *name* argument, **c:light** returns a list of all the lights defined in the drawing.

Examples

The following command lists all lights defined in the current drawing:

```
Command: (c:light "L")
("BUDLIGHT" "LIGHT01")
```

The following command lists the properties of a light named “LIGHT01”:

```
Command: (c:light "L" "LIGHT01")
("P" <Entity name: 4cf3ae8> 1.0 (26.5609 43.423 48.6995) (0.0 0.0 0.0)
(0.705882 0.705882 0.705882) 512 nil nil 3.0 "OFF" 0 nil)
```

M—Modify Lights

Modifies existing lights.

```
(c:light "M" name [intensity [from [to [color
  [shadowmapsize [hotspot [falloff [shadowsoftness
  [shadow [shadowobjects [month [day [hour [minute
  [daylight [latitude [longitude
  [attenuation]]]]]]]]]]]]]]]]]])
```

Arguments

The arguments for the Modify mode are described in the following table:

LIGHT—“M” mode arguments

Argument	Data type	Description	Default
name	STR	Unique light name	None
intensity	REAL	A real number from 0.0 to the default maximum	Based on attenuation
from	LIST	Light location	Current look-from point
to	LIST	Light target	Current look-at point
color	LIST	Any RGB triplet	1.0, 1.0, 1.0

LIGHT—“M” mode arguments (*continued*)

Argument	Data type	Description	Default
shadowmapsize	INT	Integer from 0 to 4096 (the size, in pixels, of one side of the shadow map)	0
hotspot	REAL	Angle of the brightness beam in degrees (must be in the range of 1–160)	44.0
falloff	REAL	Angle that includes the rapid decay area, in degrees (must be in the range 0–160 and greater than the hotspot value)	45.0
shadowsoftness	REAL	Real number in the range 0.0–10.0	0.0
shadow	STR	Shadow-casting toggle. Valid values are: “off” (no shadows) and “on” (cast shadows)	0.0
shadowobjects	ENAME	A selection of objects that bound the shadow maps	0.0
month	INT	Integer from 1 to 12	9
day	INT	Integer from 1 to 31	21
hour	INT	Integer from 0 to 24	15
minute	INT	Integer from 0 to 59	0
daylight	STR	Daylight savings toggle. Valid values are: “off” (no daylight savings) and “on” (daylight savings)	“off”
latitude	REAL	Real number in the range 0–90	37.62
longitude	REAL	Real number in the range 0–180	122.37
timezone	INT	Integer from –12 to 12, representing the hours behind Greenwich Mean Time (GMT)	8 (PST)
attenuation	INT	0 = no attenuation 1 = inverse linear attenuation 2 = inverse square attenuation	1

The *hotspot* and *falloff* arguments apply only to spotlights. You must pass them as `nil` when you create a new distant light.

You can specify `nil` for any argument that does not apply to the type of light you are modifying, or if you want the property affected by the argument to retain its current value. You can omit any arguments located at the end of the argument list (for example, *attenuation*, or *attenuation* and *timezone*, or *attenuation*, *timezone*, and *longitude*...).

Examples

The following code changes the color of the distant light named "D1" to blue:

```
(c:light "M" "D1" nil nil nil '(0.0 0.0 1.0))
```

ND—New Distant Light

Create a new distant light.

```
(c:light "ND" name [intensity [from [to [color
  [shadowmapsize [ nil [ nil [shadowsoftness [shadow
  [month [day [hour [minute [daylightsavings [latitude
  [longitude [timezone [attenuation
  [shadowobjects]]]]]]]]]]]]]]]]]])
```

Arguments

The arguments for the New Distant Light mode are described in the following table:

LIGHT—"ND" mode arguments

Argument	Data type	Description	Default
name	STR	Unique light name	None
intensity	REAL	A real number from 0.0 to the default maximum	Based on attenuation
from	LIST	Light location	Current look-from point
to	LIST	Light target	Current look-at point
color	LIST	Any RGB triplet	1.0, 1.0, 1.0
shadowmapsize	INT	Integer from 0 to 4096 (the size, in pixels, of one side of the shadow map)	0

LIGHT—"ND" mode arguments (*continued*)

Argument	Data type	Description	Default
hotspot	REAL	Angle of the brightness beam in degrees (must be in the range of 1–160)	44.0
falloff	REAL	Angle that includes the rapid decay area, in degrees (must be in the range 0–160 and greater than the hotspot value)	45.0
shadowsoftness	REAL	Real number in the range 0.0–10.0	0.0
shadow	STR	Shadow-casting toggle. Valid values are: "off" (no shadows) and "on" (cast shadows)	0.0
shadowobjects	ENAME	A selection of objects that bound the shadow maps	0.0
month	INT	Integer from 1 to 12	9
day	INT	Integer from 1 to 31	21
hour	INT	Integer from 0 to 24	15
minute	INT	Integer from 0 to 59	0
daylight	STR	Daylight savings toggle. Valid values are: "off" (no daylight savings) and "on" (daylight savings)	"off"
latitude	REAL	Real number in the range 0–90	37.62
longitude	REAL	Real number in the range 0–180	122.37
timezone	INT	Integer from –12 to 12, representing the hours behind Greenwich Mean Time (GMT)	8 (PST)

NP—New Point Light

Create a new point light.

```
(c:light "NP" name [intensity [from [nil [color
  [shadowmapsize [nil [nil [shadowsoftness [shadow
  [attenuation[shadowobjects ]]]]]]]]]]])
```

Arguments

The arguments for the New Point Light mode are described in the following table:

LIGHT— "NP" mode arguments

Argument	Data type	Description	Default
name	STR	Unique light name	None
intensity	REAL	A real number from 0.0 to the default maximum	Based on attenuation
from	LIST	Light location	Current look-from point
color	LIST	Any RGB triplet	1.0, 1.0, 1.0
shadowmapsize	INT	Integer from 0 to 4096 (the size, in pixels, of one side of the shadow map)	0
shadowsoftness	REAL	Real number in the range 0.0–10.0	0.0
shadow	STR	Shadow-casting toggle. Valid values are: "off" (no shadows) and "on" (cast shadows)	0.0
attenuation	INT	0 = no attenuation 1 = inverse linear attenuation 2 = inverse square attenuation	1
shadowobjects	ENAME	A selection of objects that bound the shadow maps	0.0

Three arguments—*`to`* (after *`from`*), *`hotspot`*, and *`falloff`* (after *`shadowmapsize`*)—do not apply to point lights. You must pass them as `nil` when you create a new point light.

Examples

For example, the following code creates a new point light named NEWPT1.

```
(c:light "NP" "NEWPT1")
```

NEWPT1 would have the default intensity, the current attenuation setting, the default location looking at the current view, and the default color of white.

NOTE For point lights, the default maximum intensity depends on the current point/spotlight attenuation setting. With no attenuation, it is 1.00; with inverse linear attenuation, it is twice the drawing's maximum extents distance; and with inverse square attenuation, it is the square of twice the maximum extents distance.

NS—New Spotlight

Creates a new spotlight.

```
(c:light "NS" name [intensity [from [to [color
  [shadowmapsize [hotspot [falloff [shadowsoftness
  [shadow [attenuation [shadowobjects]]]]]]]]]]])
```

Arguments

The arguments for the New Spotlight mode are described in the following table:

LIGHT— "NS" mode arguments

Argument	Data type	Description	Default
name	STR	Unique light name	None
intensity	REAL	A real number from 0.0 to the default maximum	Based on attenuation
from	LIST	Light location	Current look-from point
to	LIST	Light target	Current look-at point
color	LIST	Any RGB triplet	1.0, 1.0, 1.0
shadowmapsize	INT	Integer from 0 to 4096 (the size, in pixels, of one side of the shadow map)	0
hotspot	REAL	Angle of the brightness beam in degrees (must be in the range of 1–160)	44.0
falloff	REAL	Angle that includes the rapid decay area, in degrees (must be in the range 0–160 and greater than the hotspot value)	45.0

LIGHT— “NS” mode arguments (*continued*)

Argument	Data type	Description	Default
shadowsoftness	REAL	Real number in the range 0.0–10.0	0.0
shadow	STR	Shadow-casting toggle. Valid values are: “off” (no shadows) and “on” (cast shadows)	0.0
attenuation	INT	0 = no attenuation 1 = inverse linear attenuation 2 = inverse square attenuation	1
shadowobjects	ENAME	A selection of objects that bound the shadow maps	0.0

Examples

The following code creates a new spotlight named “GSPOT”:

```
Command: (c:light "NS" "GSPOT" 43.82 '(12.0 6.0 24.0) '(78.0 78.0
24.0) nil nil 30.0 32.0)
1
```

GSPOT is a spotlight with an intensity of 43.82. Its color is the default (white). The spotlight’s location is (12,6,24), and its target is (78,78,24). Its cone is 32 degrees wide, with a hotspot of 30 degrees.

NOTE For spotlights, the default maximum intensity depends on the current point/spotlight attenuation setting. With no attenuation, it is 1.00; with inverse linear attenuation, it is twice the drawing's maximum extents distance; and with inverse square attenuation, it is the square of twice the maximum extents distance.

R—Rename Light

Rename a light.

(c:light "R" *old_name new_name*)

Arguments

old_name	A string specifying the name of the light to rename.
new_name	A string specifying the light’s new name.

The following function call changes the light named “GSPOT” to “HOTSPOT”:

Examples

```
Command: (c:light "R" "GSPOT" "HOTSPOT")
1
```

lsedit

Creates or modifies landscape objects (Externally-defined: *render* ARX application)

`(c:lsedit mode [options])`

This form of the **`c:lsedit`** is used to create or modify instances of landscape objects in the drawing.

`(c:lsedit "LIST" object)`

This form of **`c:lsedit`** lists the attributes of the specified landscape object. The list returned identifies the name, height, position, and view alignment of the specified object.

`(c:lsedit object height [position [alignment]])`

This form of **`lsedit`** modifies a landscape object.

Arguments

The arguments for the LSEDIT command are described in the following table:

LSEDIT arguments

Argument	Data type	Description	Default
object	ENAME	Handle of the landscape object	None
height	REAL	Height of the object in drawing units. If nil, the current value is unchanged.	None
position	LIST (of reals)	The position of the base of the object. If nil, the current value is unchanged.	None

LSEDIT arguments (*continued*)

Argument	Data type	Description	Default
alignment	INT	Specifies the geometry and alignment of the entry: 0—view-aligned single face 1—non-view-aligned single face 2—non-view-aligned crossing faces 3—view-aligned crossing faces If nil, the current value is unchanged.	None

Examples

Modify a landscape object, where <ename> is the AutoCAD name (entsel) of the object to modify; leave alignment unchanged:

```
(c:lsedit <ename> 35.0 '(10.0 23.0) nil)
```

lslib

Manages the landscape library (Externally-defined: *render* ARX application)

(c:lslib mode *[options]*)

Arguments

mode	The *mode* arguments can be one of the following: **ADD** Add an entry to a landscape library **DELETE** Delete an entry from a landscape library **MODIFY** Modify an entry in a landscape library **OPEN** Open a landscape library **SAVE** Save the current landscape library **LIST** List the entries in the current landscape library
options	The allowable *options* arguments varies depending on *mode*. See the description of each mode to determine the allowable options.

ADD

Add an entry to the current library.

`(c:lslib "ADD"` ***`name texture-map opacity-map alignment`*****`)`**

Arguments

name	A string naming the entry in the landscape library.
texture-map	A string naming the image file for the entry.
opacity-map	A string naming the opacity image for the entry.
alignment	An integer specifying the geometry and alignment of the entry. Can be one of the following:

0	view-aligned single face
1	non-view-aligned single face
2	non-view-aligned crossing faces
3	view-aligned crossing faces

There are no default values for any of these arguments.

Examples

Add an entry called “Maple tree” to the current landscape library:

```
(c:lslib "ADD" "Maple tree" "maple.tga" "mapleo.tga" 0)
```

DELETE

Remove an entry from the current library.

`(c:lslib "DELETE"` ***`name`*****`)`**

Arguments

name	A string naming the entry in the landscape library.

Examples

Remove the entry called “Maple tree” from the current landscape library:

```
(c:lslib "delete" "Maple tree")
```

MODIFY

(c:lslib "MODIFY" ***name texture-map [opacity-map [alignment]]*****)**

Change an entry in the current library. The *texture-map*, *opacity-map*, and *alignment* arguments can be passed as `nil`, in which case the value is unchanged.

Arguments

name	A string naming the entry in the landscape library.
texture-map	A string naming the image file for the entry.
opacity-map	A string naming the opacity image for the entry.
alignment	An integer specifying the geometry and alignment of the entry. Can be one of the following:
	0 view-aligned single face
	1 non-view-aligned single face
	2 non-view-aligned crossing faces
	3 view-aligned crossing faces

There are no default values for any of these arguments.

Examples

Change the "Maple tree" to be non-view-aligned with crossing faces:

```
(c:lslib "MODIFY" "Maple tree" nil nil 2)
```

OPEN

Open a new library and make it the current library.

(c:lslib "OPEN" ***name*****)**

Arguments

name	A string naming the landscape library to open.

Examples

Open the *TREES.LLI* file and make it the current landscape library:

```
(c:lslib "OPEN" "TREES.LLI")
```

SAVE

Save the current landscape library as the named file.

(c:lslib "SAVE" *name*)

Arguments

name	A string naming the landscape library file.

Examples

Save the file as *TREES.LLI*:

```
(c:lslib "SAVE" "TREES.LLI")
```

LIST

Lists all the elements in the current library. This command takes no arguments. The list includes landscape entries of the form '("NAME" "TEX-MAP" "OP-MAP" ALIGN).

(c:lslib "LIST")

Examples

The following illustrates output from the LIST option:

```
(("Bush #1" "8bush02l.tga" "8bush02o.tga" 0)
("Cactus" "8plnt15l.tga" "8plnt15o.tga" 0)
("Dawn Redwood" "8tree39l.tga" "8tree39o.tga" 0))
```

lsnew

Create landscape objects (Externally-defined: *render* ARX application)

(c:lsnew *object-type height position alignment*)

The LSNEW command is used to create instances of landscape objects in the drawing.

Arguments

object-type	A string naming the landscape library entry.
height	A real number indicating the height of the object in drawing units.
position	A list of reals indicating the position of the base of the object.

alignment	An integer specifying the geometry and alignment of the entry. Can be one of the following: **0** view-aligned single face **1** non-view-aligned single face **2** non-view-aligned crossing faces **3** view-aligned crossing faces

There are no default values for any of these arguments.

Examples

Create a new instance of “Cactus” that is 25 units tall, located at 0, 1, 3, and has a single non-view-aligned face.

```
Command: (c:lsnew "Cactus" 25.0 '(0.0 1.0 3.0) 1)
1
```

matlib

Manages materials libraries (Externally-defined: *render* ARX application)

(c:matlib *mode name [file]*)

Arguments

mode	A string that specifies the action that this function performs. Can be one of the following: **I** Import a material from a library. **E** Export a material to a library. **D** Delete a material from the drawing. **C** Delete unattached materials from the drawing. **L** List materials
name	A string that specifies the name of the material to import, export, or delete.
file	A string that specifies the name of the materials library. file. The `file` argument must include the *.mli* extension

Examples

Imports the material BRASS from the standard AutoCAD Render materials library, *render.mli*:

```
Command: (c:matlib "I" "brass" "c:/acad2000/support/render.mli")
1
```

The *`file`* argument is not used with the Delete mode:

```
(c:matlib "D" "steel")
```

mirror3d

Reflects selected objects about a user-specified plane (Externally-defined: *geom3d* ARX application)

```
(mirror3d arg1 arg2 ...)
```

Arguments

The order, number, and type of arguments for the **`mirror3d`** function are the same as if you were entering the MIRROR3D AutoCAD command. To signify a user pressing ENTER without typing any values, use `nil` or an empty string (`""`).

Return Values

`T` if successful, otherwise `nil`.

Examples

The following example mirrors the selected objects about the *XY* plane that passes through the point 0,0,5, and then deletes the old objects:

```
(setq ss (ssget))
(mirror3d ss "XY" '(0 0 5) "Y")
```

render

Creates a realistically shaded image of a 3D wireframe model using geometry, lighting, and surface finish information (Externally-defined: *render* ARX application)

(c:render *[filename|point1 point2]*)

Arguments

filename	A string naming a rendering file. If the *filename* argument is present, the rendering is written to a file of that name. If a driver to render to a file hasn't been configured, the *filename* argument is ignored. The current configuration must specify rendering to a file.
point1	A list of reals indicating the first crop window point.
point1	A list of reals indicating the second crop window point.

The rendering is controlled by the current settings; set these by using the **c:rpref** function. For example:

```
(c:rpref "Toggle" "CropWindow" "On")
```

NOTE When the current rendering preferences specify *Query for Selection* and the PICKFIRST system variable is turned on, then if a selection set is current when you invoke **c:render**, the objects in the set are rendered with no further prompting.

Setting the Render to File Options

Sets the render to file options for rendering.

**(c:rfileopt *fileformat xres yres aratio colormode*
  <mode-specific options>)**

Arguments

The following table describes the **`c:rfileopt`** arguments.

RFILEOPT arguments

Argument	Data type	Description
fileformat	STR	Identifier for the requested format: TGA—Targa format PCX—Z-Soft bitmap format BMP—Microsoft Windows format PS—PostScript TIFF—Tagged Image File Format
xres	INT	X resolution of the output file (valid values range from 1 to 4096)
yres	INT	Y resolution of the output file (valid values range from 1 to 4096)
aratio	REAL	Pixel aspect ratio
colormode	STR	Each file format accepts a subset of the following values: MONO—Monochrome G8—256 gray levels C8—256 colors C16—16-bit color C24—24-bit color C32—24-bit color with 8 bits of alpha

TGA

Specifies the Targa format.

```
(c:rfileopt "TGA" xres yres aratio colormode
  interlace compress bottomup)
```

Arguments

TGA format arguments

Argument	Data type	Description
colormode	STR	Color mode: G8, C8, C24, or C32
interlace	INT	Interlace mode: 1—no interlace 2—2:1 interlace 4—4:1 interlace
compress	STR	Compression (default = "COMP"): COMP—Compression on nil—No compression
bottomup	STR	Bottom up (default = "UP"): UP—bottom up nil—top down

Examples

```
(C:RFILEOPT "TGA" 640 480 1.0 "C32" 1 "COMP" "UP")
```

PCX

Specifies the Z-Soft Bitmap format.

```
(c:rfileopt "PCX" xres yres aratio colormode)
```

Arguments

PCX format arguments

Argument	Data type	Description
colormode	STR	Color mode: MONO, G8, or C8

Examples

```
(C:RFILEOPT "PCX" 640 480 1.0 "G8")
```

BMP

Specifies the Microsoft Windows bitmap format.

(c:rfileopt "BMP" *xres yres aratio colormode*)

Arguments

BMP format arguments

Argument	Data type	Description
colormode	STR	Color mode: MONO, G8, or C8

Examples

```
(C:RFILEOPT "BMP" 640 480 1.0 "C8")
```

PS

Specifies the PostScript format.

(c:rfileopt "PS" *xres yres aratio colormode portrait imagesize [size]*)

Arguments

PS format arguments

Argument	Data type	Description
colormode	STR	Color mode: MONO, G8, C8, or C24
portrait	STR	Landscape or portrait (default = "L"): P—Portrait L—Landscape
imagesize	STR	Type (default = "A") A—Auto I—Image C—Custom
size	INT	Size of the image

Examples

```
(C:RFILEOPT "PS" 640 480 1.0 "C24" "P" "C" 640)
```

TIFF

Specifies the Tagged Image File format.

`(c:rfileopt "TIFF"` ***`xres yres aratio colormode`*****`)`**

Arguments

TIFF format arguments

Argument	Data type	Description
colormode	STR	Color mode: MONO, G8, C8, C24, or C32

Examples

```
(C:RFILEOPT "TIFF" 640 480 1.0 "C24")
```

renderupdate

Regenerate the ent2face file on the next rendering (Externally-defined: *render* ARX application)

`(c:renderupdate [`*`RU_value`*`])`

Use the **`renderupdate`** command with no arguments to regenerate the *en2face* file on the next rendering.

Arguments

RU_value	A string specifying one of the following: **ALWAYS** Generate a new geometry file for each rendering. **OFF** Return Render to the normal geometry caching mode.

replay

Displays a BMP, TGA, or TIFF image (Externally-defined: *render* ARX application)

(c:replay *filename type [xoff yoff xsize ysize]*)

With the REPLAY command, you can display BMP, TGA, or TIFF files on the AutoCAD rendering display. Use this command's function to replay the image file at various offsets and sizes.

Arguments

filename	A string naming the image file.
type	A string identifying the file type. Can be BMP, TGA, or TIFF.
xoff	An integer specifying the image *X* offset in pixels. Default is 0.
yoff	An integer specifying the image *Y* offset in pixels. Default is 0.
xsize	Image *X* size in pixels. Default is the actual *X* size.
ysize	Image *Y* size in pixels. Default is the actual *Y* size.

Examples

The following call replays an image named *test.tga*, displaying pixels starting from the lower left of the image (zero offset) out to 500 pixels wide and 400 pixels in height:

```
(c:replay "TEST" "TGA" 0 0 500 400)
```

rmat

Creates, edits, attaches, and detaches rendering materials (Externally-defined: *render* ARX application)

```
(c:rmat mode options)
```

Arguments

mode	A string. Can be one of the following: **A** Attaches material **C** Copies material **D** Detaches material **L** Lists all materials in the drawing or returns a definition of the specified material **M** Modifies material **N** Creates new material
options	The options allowed depend on the *mode* specified.

A—Attach Material

The "A" (attach) mode lets you attach a material to selected objects or an ACI (AutoCAD Color Index) value, depending on whether the third argument (layer-name) is an integer or a selection set.

```
(c:rmat "A" name [aci | selection-set | layer-name])
```

Arguments

The following table describes the attach arguments.

Attach arguments

Argument	Data type	Description
name	STR	Name of the material to attach
aci	INT	ACI number in the range of 0 through 255
selection-set	INT	Selection set that contains the entities to attach
layer-name	STR	Name of the layer

Examples

Attach the material PURPLE TIGER to the ACI 1 (red):

```
(c:rmat "A" "PURPLE TIGER" 1)
```

If you omit the third argument, the "A" mode returns a list of three items:

- A list of layer names the material is attached to.
- A list of ACIs the material is attached to.
- A selection set that contains the objects the material is attached to.

The following example illustrates the values returned when the third argument is omitted:

```
Command:  (c:rmat "a" "twood")
Gathering objects…1 found
                        Layer names ACI's
(("first" "second")(135) <Selection set 12>))
```

A material index value in the range 1–255 is an ACI number; an index greater than 255 indicates an AutoCAD Render material not assigned by ACI.

C—Copy Material

Creates a new material by copying one already present in the drawing.

(c:rmat "C" *cur_name new_name*)

Arguments

cur_name	A string that specifies the name of the material to copy.
new_name	A string that specifies the name for the new material.

Examples

Modify a material to change its definition:

```
(c:rmat "C" "RED" "RED2")
```

D—Detach Material

The "D" (detach) mode lets you detach a material from selected objects, an ACI (AutoCAD Color Index) value, or layers, depending on whether the second argument (*selection-set*) is an integer, a selection set, or a string.

(c:rmat "D" *name [aci | selection-set | layer-name]*)

Arguments

The following table describes the detach arguments.

Detach arguments

Argument	Data type	Description	Default
name	STR	Name of the material to detach	None
aci	INT	ACI number in the range of 0 through 255	None
selection-set	INT	Selection set that contains the entities to detach	None
layer-name	STR	Name of the layer	None

Examples

Prompt the user to select objects, and then detach each object from its material:

```
(c:rmat "D" (ssget))
```

L—List Material

Lists material definitions in the drawing.

`(c:rmat "L"` ***`[name]`*****`)`**

Arguments

name	A string that specifies the material definition to list. If the *`name`* argument is omitted, **`c:rmat`** lists all materials in the drawing.

Examples

List all materials in the drawing:

```
Command:  (c:rmat "L")
("*GLOBAL*" "BLUE GLASS" "WHITE PLASTIC" "TWOOD" "BEIGE MATTE")
```

The first string in the list specifies the default global material, *GLOBAL*. You can pass this string to **`c:rmat`** just as you can pass the names of library or user-defined materials, as demonstrated in the following example:

```
Command:  (c:rmat "L" "*GLOBAL*")
("*GLOBAL*" "STANDARD" (-1.0 -1.0 -1.0) 0.7 ("" 0.0 0 (1.0 1.0) (0.0
0.0) 0.0 0 0) (-1.0 -1.0 -1.0) 0.1 (-1.0 -1.0 -1.0) 0.2 ("" 0.0 0)
0.5 0.0 ("" 0.0 0 (1.0 1.0) (0.0 0.0) 0.0 0 0) 1.0 ("" 0.0 0 (1.0
1.0) (0.0 0.0) 0.0 0 0))
```

The list items in a material definition are the same as the arguments to the Modify or New modes.

M—Modify Material

The options for the "M" (modify) mode are the same as for the "N" (new) mode. If an argument is `nil`, or is omitted from the end of the argument list, the property affected by the argument retains its current value.

For example, the following call changes BLUE MARBLE to have a medium blue stone (matrix) color and black veins:

```
(c:rmat "M" "BLUE MARBLE" "marble" '(0.5 0.5 1.0) '(0.0 0.0 0.0))
```

N—New Material

The "N" (new) mode creates a new material. The arguments to this function depend not only on the mode, but also on the type of material you're creating. The procedural materials: marble, granite, and wood, each have a unique set of arguments, which differs from the standard material arguments.

Arguments

The following table describes the new arguments:

New arguments

Argument	Data type	Description	Default
name	STR	Name of the material to create	None
material-type	STR	Type of new material. The options are: STANDARD—standard material MARBLE—marble material GRANITE—granite material WOOD—wood material	None
description	(Varies)	Arguments depend on the type of material you're creating	(Varies)
selection-set	INT	Selection set that contains the entities to detach	None
layer-name	STR	Name of the layer	None

In addition, the arguments for each kind of bitmap are specified in a sublist as described under “Bitmap Arguments” on page 407.

Standard

The material type string “STANDARD” indicates you’re creating a new standard material.

```
(c:rmat "N" name "STANDARD" [color [color-weight [pattern
  [ambient [amb-weight [refl [refl-weight [refl-map
  [roughness [transparency [opacitymap [refraction
  [bumpmap]]]]]]]]]]]]])
```

Arguments

The following table describes the standard arguments:

Standard arguments

Argument	Data type	Description	Default
color	LIST (or reals)	Material color as an RGB triple; (–1.0 – 1.0 –1.0) means derive the color from an object’s ACI (diffuse color)	(–1.0 –1.0 –1.0)— By ACI
color-weight	REAL	Weight factor (color Value)—the amount of diffuse color	0.7
pattern	LIST	Pattern/texture map arguments	None
ambient	LIST (or reals)	Ambient (shadow) color as an RGB triple	(–1.0 –1.0 –1.0)— By ACI
amb-weight	REAL	Weight factor (ambient Value)—the amount of specular color	0.1
refl	LIST (or reals)	Reflection (specular) color as an RGB triple	(–1.0 –1.0 –1.0)— By ACI
refl-weight	REAL	Weight factor (reflection Value)— the amount of specular color	0.2
refl-map	LIST	Reflection/environment map arguments	None

Standard arguments (*continued*)

Argument	Data type	Description	Default
roughness	REAL	Roughness—the size of a specular highlight	0.5
transparency	REAL	Transparency of the material	0.0
opacity-map	LIST	Opacity map arguments	None
refraction	REAL	Index of refraction	1.0
bumpmap	LIST	Bump map arguments	None

Examples

The following call creates a shiny red material with a pattern map:

```
(c:rmat "N" "RED LACQUER" "STANDARD" ; Name and type
'(1.0 0.0 0.0) (1.0) ; Color (red), weight, and texture map
'("INLAY.TGA" 0.75 0 (0.5 0.5) (0.3 0.3) 0.0 0 1)
'(1.0 0.0 0.0) 1.0   ; Ambient color and its weight (same as diffuse)
'(1.0 0.0 0.0) 1.0    ; Reflection color (white) and its weight
nil                   ; No reflection map
0.2                   ; Roughness (low)
0.0                   ; Transparency (none)
nil                   ; No opacity map
0.0                   ; Refraction (none)
nil                   ; No bump map
```

The next call creates a material, MAPS, that uses multiple bitmaps:

```
(c:rmat "N" "MAPS" "STANDARD"
'(1.0 0.0 0.0) (1.0) '("weave.tga" 1.0 0)
'(1.0 0.0 0.0) 1.0
'(1.0 0.0 0.0) 1.0 '("room.tga" 0.75)
0.5
0.0
'("hole.tga")
1.0
'("ridges.tga")
```

The following call creates a material with no bitmaps and default values, with reflections that are generated by ray tracing when rendered with Photo Raytrace or with environment map with Photo Real:

```
(c:rmat "N" "SHINE" "STANDARD" nil nil nil nil nil nil nil
'(nil nil 1))
```

Marble

The material type string “MARBLE” indicates that you are creating a new marble material.

```
(c:rmat "N" name "MARBLE" [stone-color [vein-color
  [refl [refl-weight [refl-map [roughness [turbulence
  [sharpness [scale [bumpmap ]]]]]]]]]])
```

Arguments

The following table describes the marble arguments:

RMAT—Marble arguments

Argument	Data type	Description	Default
stone-color	LIST (of reals)	RGB value specifying the main matrix color of the marble	(–1.0 –1.0 –1.0)—white
vein-color	LIST (of reals)	RGB value specifying the vein color of the marble	(–1.0 –1.0 –1.0)—black
refl	LIST (of reals)	Reflection (specular) color as an RGB value	(–1.0 –1.0 –1.0)—By ACI
refl-wgt	REAL	Weight factor (reflection Value)—the amount of specular color	0.2
refl-map	LIST	Reflection/environment map arguments	None
roughness	REAL	Roughness—the size of a specular highlight	0.5
turbulence	INT	Turbulence factor—swirliness of the veins	3
sharpness	REAL	Sharpness factor—the amount of blur	1.0
scale	REAL	Overall scale factor	0.16
bumpmap	LIST	Bumpmap arguments	None

Examples

The following call creates a marble with a pink matrix and black veins:

```
(c:rmat "N" "PINK MARBLE" "MARBLE" '(1.0 0.34 0.79))
```

Granite

The material type string "GRANITE" indicates that you're creating a new granite material.

```
(c:rmat "N" name "GRANITE" [first-color [amount1
  [second-color [amount2 [third-color [amount3
  [fourth-color [amount 4 [refl [refl-weight
  [refl-map [roughness [sharpness [scale
  [bumpmap ]]]]]]]]]]]]]]])
```

Arguments

The following table describes the granite arguments:

RMAT—Granite arguments

Argument	Data type	Description	Default
first-color	LIST (of reals)	RGB value	(–1.0 –1.0 –1.0)—white
amount1	REAL	Weight factor (color Value) for first color	1.0
second-color	LIST (of reals)	RGB value	(0.5 0.5 0.5)—dark gray
amount2	REAL	Weight factor (color Value) for second color	1.0
third-color	LIST (of reals)	RGB value	(0.0 0.0 0.0)—black
amount3	REAL	Weight factor (color Value) for third color	1.0
fourth-color	LIST (of reals)	RGB value	(0.7 0.7 0.7)—light gray
amount4	REAL	Weight factor (color Value) for fourth color	1.0
refl	LIST (of reals)	Reflection (specular) color as an RGB value	(–1.0 –1.0 –1.0)—By ACI
refl-weight	REAL	Weight factor (reflection Value)—the amount of specular color	0.2

RMAT—Granite arguments (*continued*)

Argument	Data type	Description	Default
refl-map	LIST	Reflection/environment map arguments	None
roughness	REAL	Roughness—the size of a specular highlight	0.5
sharpness	REAL	Sharpness factor—the amount of blur	1.0
scale	REAL	Overall scale factor	0.16
bumpmap	LIST	Bumpmap arguments	None

Examples

Create a granite without dark gray, with more black, and with yellow instead of light gray:

```
(c:rmap "N" "YELLOW GRANITE"
 nil 0.5 nil 0.0 nil 0.85 '(1.0 1.0 0.0) 0.6)
```

Wood

The material type string “WOOD” indicates that you’re creating a new wood material.

```
(c:rmat "N" name "WOOD" [light-color
  [dark-color [refl [refl-weight [refl-map [roughness
  [ratio [density [width [shape [bumpmap ]]]]]]]]]]])
```

Arguments

The following table describes the wood arguments:

RMAT—Wood arguments

Argument	Data type	Description	Default
light-color	LIST (of reals)	RGB value specifying the color of the light rings	(0.6 0.4 0.3)
dark-color	LIST (of reals)	RGB value specifying the color of the dark rings	(0.3 0.2 0.2)—black

RMAT—Wood arguments (*continued*)

Argument	Data type	Description	Default
refl	LIST (of reals)	Reflection (specular) color as an RGB value	(–1.0 –1.0 –1.0)—By ACI
refl-weight	REAL	Weight factor (reflection Value)—the amount of specular color	0.2
refl-map	LIST	Reflection/environment map arguments	None
roughness	REAL	Roughness—the size of a specular highlight	0.5
ratio	REAL	Ratio of light to dark rings	0.5
density	REAL	Density of the rings	6.0
width	REAL	Ring width variation	0.2
shape	REAL	Ring shape variation	0.2
scale	REAL	Overall scale factor	0.16
bumpmap	LIST	Bumpmap arguments	None

Examples

Create a wood with an irregular grain:

```
(c:rmat "N" "CRYPTO" "WOOD" nil nil nil nil nil nil nil nil nil 0.56)
```

Bitmap Arguments

The arguments to specify a bitmap are passed to a list, which you can include as a sublist in the **`c:rmat`** call (this is the form shown at the beginning of each of the following sections) or assign to a symbol before you call **`c:rmat`**.

Pattern/Texture

```
'(name [blend [repeat [scale [offset [reserved [map-style
   [auto-axis]]]]]]])
```

Arguments

The following table describes the pattern/texture arguments:

Pattern/texture arguments

Argument	Data type	Description	Default
name	STR	Name of the bitmap file	None
blend	REAL	Amount of map color to use	1.0
repeat	INT	Whether to repeat (tile) the bitmap: 0—no tiling (crop) 1—tile (repeat pattern)	0
scale	LIST (of reals)	U and V scale factors	(1.0 1.0)
offset	LIST (of reals)	U and V offsets	(0.0 0.0)
reserved	REAL	Reserved placeholder	None
map-style	INT	Whether the map style is: 0—fixed scale 1—fit to entity	0
auto-axis	INT	Whether or not Auto Axis is enabled: 0—disabled 1—enabled	1

Reflection/Environment

```
'(name [blend [raytrace]])
```

Arguments

The following table describes the reflection/environment arguments:

Reflection/environment arguments

Argument	Data type	Description	Default
name	STR	Name of the bitmap file	None
blend	REAL	Amount of map color to use	1.0
mirror	REAL	Whether to generate mirrored reflections: 0—no mirror 1—mirror During mirror generates raytraced reflections; during scanline uses environment map for reflections	0

Opacity

```
'(name [blend [repeat [scale [offset [reserved [map-style
   [auto-axis]]]]]]])
```

Arguments

The following table describes the opacity arguments:

Opacity arguments

Argument	Data type	Description	Default
name	STR	Name of the bitmap file	None
blend	REAL	Amount of map color to use	1.0
repeat	INT	Whether to repeat (tile) the bitmap: 0—no tiling (crop) 1—tile (repeat pattern)	0

Opacity arguments (*continued*)

Argument	Data type	Description	Default
scale	LIST (of reals)	U and V scale factors	(1.0 1.0)
offset	LIST (of reals)	U and V offsets	(0.0 0.0)
reserved	REAL	Reserved placeholder	None
map-style	INT	Whether the map style is: 0—fixed scale 1—fit to entity	0
auto-axis	INT	Whether or not Auto Axis is enabled: 0—disabled 1—enabled	1

Bump Map

```
'(name [amplitude [repeat [scale [offset [reserved
  [map-style [auto-axis]]]]]]])
```

The following table describes the bump arguments:

Bump arguments

Argument	Data type	Description	Default
name	STR	Name of the bitmap file	None
amplitude	REAL	Degree of bumpiness	1.0
repeat	INT	Whether to repeat (tile) the bitmap: 0—no tiling (crop) 1—tile (repeat pattern)	0
scale	LIST (of reals)	U and V scale factors	(1.0 1.0)
offset	LIST (of reals)	U and V offsets	(0.0 0.0)
reserved	REAL	Reserved placeholder	None

Bump arguments (*continued*)

Argument	Data type	Description	Default
map-style	INT	Whether the map style is: 0—fixed scale 1—fit to entity	0
auto-axis	INT	Whether or not Auto Axis is enabled: 0—disabled 1—enabled	1

rotate3d

Rotates an object about an arbitrary 3D axis (Externally-defined: *geom3d* ARX application)

`(rotate3d` *`args`* `...)`

Arguments

args The order, number, and type of arguments for the **`rotate3d`** function are the same as if you were entering them at the command line; see ROTATE3D in the *Command Reference* for more information.

To signify a null response (user pressing ENTER without specifying any arguments), use `nil` or an empty string (`""`).

Return Values

If successful, **`rotate3d`** returns `T`; otherwise it returns `nil`.

Examples

The following example rotates the selected objects 30 degrees about the axis specified by points *`p1`* and *`p2`*.

```
(setq ss (ssget))
(rotate3d ss p1 p2 30)
```

AutoLISP support for the **`rotate3d`** function is implemented with the use of the SAGET library.

rpref

Sets rendering preferences (Externally-defined: *render* ARX application)

(c:rpref *mode option [setting]*)

THE **c:rpref** FUNCTION determines which rendering parameters will be used, and which rendering behavior will be the default.

Arguments

mode	A string that can be one of the following: **DEST** Destination of viewport, Render window, or file **ICON** Scale of the Light and Materials icon blocks **ROPT** More rendering options **SELECT** Whether to prompt for object selection **STYPE** Rendering type of Render, Photo Real, or Photo Raytrace **TOGGLE** Rendering options
option	Depends on *mode*.
setting	Depends on *mode*.

See Also "Setting the Render to File Options" on page 392**.**

DEST—Destination Preference

Selects which output device is used.

(c:rpref "DEST" *option*)

Arguments

option	A string that specifies the rendering destination. Can be one of the following: **FRAMEBUFFER** Render to display **HARDCOPY** Render to Render window **FILE** Render to file

Examples

The following call specifies rendering to a file:

```
(c:rpref "DEST" "FILE")
```

ICON—Icon Preference

Specifies the size of the light or material icon block in a drawing.

(c:rpref "ICON" ***option*****)**

Arguments

option	A real that specifies the size of the icon block. The default value is 1.00.

Examples

The following function call changes the icon scale to 50 percent:

```
(c:rpref "ICON" 0.5)
```

STYPE—Rendering Type Preference

Specifies which type of Render is used.

(c:rpref "STYPE" ***option*****)**

Arguments

option A string that specifies the rendering type. Can be one of the following:

ARENDER	Basic rendering
ASCAN	Photo Real rendering
ARAY	Photo Raytrace rendering

Examples

The following code specifies that the next rendering will be generated by the basic AutoCAD renderer.

```
(c:rpref "STYPE" "ARENDER")
```

SELECT—Selection Preference

Specifies whether to prompt for object selection before generating a rendering.

(c:rpref "SELECT" ***option*****)**

Arguments

option	A string that specifies the prompting. Can be one of the following: **ALL** Render full scene **ASK** Prompt for object selection

Examples

The following call sets rendering to prompt for object selection:

```
(c:rpref "SELECT" "ASK")
```

TOGGLE—Toggle Preference

Controls various rendering options.

`(c:rpref "TOGGLE"` ***`option setting`*****`)`**

Arguments

option	A string that specifies the prompting. Can be one of the following: **CACHE** Render to a cache file. As long as the drawing geometry or view is unchanged, the cached file is used for subsequent renderings, eliminating the need to retessellate. **SHADOW** Render with shadows. **SMOOTH** Render with smoothing. **MERGE** Merge objects with background. **FINISH** Apply materials. **SKIPRDLG** Do not display the Render dialog box.
setting	A string that specifies the state of the toggle. Possible values for *`setting`* are `"ON"` and `"OFF"`.

Examples

The following calls turn off Merge rendering and turn on shadows:

```
(c:rpref "TOGGLE" "MERGE" "OFF")
(c:rpref "TOGGLE" "SMOOTH" "ON")
```

saveimg

Saves a rendered image to a file in BMP, TGA, or TIFF format (Externally-defined: *render* ARX application)

```
(c:saveimg filename type [portion] [xoff yoff xsize
  ysize] [compression])
```

When AutoCAD is configured to render to a separate display, the `portion` argument should not be used. You can specify a size and offset for the image; and for TGA and TIFF files, you can specify a compression scheme.

Arguments

The arguments to **`saveimg`** are described in the following table:

SAVEIMG function arguments

Argument	Data type	Description	Default
filename	STR	Image file name	None
type	STR	File type: BMP, TGA, or TIFF	None
portion	STR	Portion of the screen to save: A—active viewport D—drawing area F—full screen **NOTE** This argument is now ignored, but is provided for script compatibility.	"A"
xoff	INT	*X* offset in pixels	0
yoff	INT	*Y* offset in pixels	0
xsize	INT	*X* size in pixels	Actual *X* size
ysize	INT	*Y* size in pixels	Actual *Y* size
compression	STR	Compression scheme: NONE PACK (TIFF files only) RLE (TGA files only)	None

Examples

The following example saves a full-screen TIFF image named *test.tif*, without compressing the file:

```
(c:saveimg "TEST" "TIF" "NONE")
```

scene

Creates new scenes and modifies or deletes existing scenes in paper space only (Externally-defined: *render* ARX application)

`(c:scene mode [options])`

Arguments

mode	A string that can be one of the following:
	D Deletes an existing scene
	L Lists all scenes in the drawing or returns a definition of the specified scene
	M Modifies an existing scene
	N Creates a new scene
	R Renames an existing scene
	S Sets the current scene
options	The allowable options depend on the *mode* specified.

D—Delete Scene

Deletes an existing scene.

`(c:scene "D" name)`

Arguments

name	A string that specifies the name of the scene to delete.

If the deleted scene is the current scene, *NONE* becomes the current scene.

Examples

```
(c:scene "D" "PLANVIEW")
```

L—List Scene

Lists all scenes in the drawing or returns a definition of the specified scene.

(c:scene "L" ***[name]*****)**

Arguments

name	A string that specifies the name of the scene to list. If the *name* argument is omitted, **c:scene** returns a list of all the scenes defined in the drawing.

Return Values

When you specify *name*, **c:scene** returns the definition of the named scene.

Examples

The following code returns a list of scene names defined in the drawing.

```
Command: (c:scene "L")
("" "SCENE1" "SCENE2" "SCENE3")
```

The empty string ("") is the default scene, *NONE*, which can't be modified.

The following function call returns a definition of the named scene:

```
Command: (c:scene "L" "SCENE2")
(T T)
("VIEW1" nil)
("VIEW2" ("LIGHT1" "LIGHT2"))
```

M—Modify Scene

Modifies an existing scene.

(c:scene "M" ***name [view [lights]]*****)**

The options for the Modify mode are the same as those for the New mode, except that you can pass *view* as `nil` to modify only the lights.

NOTE You must pass the *lights* argument as a list even when you specify only one light.

For example, the following call modifies a scene named SCENE1 to use the named view FRONT and all the lights in the drawing:

```
(c:scene "M" "SCENE1" "FRONT" (C:LIGHT "L"))
```

The following call modifies SCENE1 to use the named view BACK and only the lights P1 and P2:

```
(c:scene "M" "SCENE1" "BACK" '("P1" "P2"))
```

N—New Scene

Creates a new scene.

`(c:scene "N"` ***`name [view [lights]]`*** **`)`**

Arguments

name	A string that specifies the name of the new scene.
view	Either a string identifying an AutoCAD named view, or the symbol `T` to indicate *CURRENT* view.
lights	The *lights* argument can be one of the following:

- A list of strings containing light names to be used.
- The symbol `T`, indicating *ALL* lights in the drawing.
- `Nil`, indicating no lights in the drawing.

SCENE—“N” mode argument

Argument	Data type	Description	Default
lights	LIST (of strings)	List of light names. Must use a list even if specifying a single light.	*ALL* lights in the drawing
	T (SYM)	Use *ALL* lights in the drawing	
	nil	Use no lights in the drawing	An “over-the-shoulder” distant light

Examples

To create a new scene named DEFAULT using the *CURRENT* view and *ALL* lights, issue the following function call:

```
(c:scene "N" "DEFAULT")
```

To create a new scene named DULL using the *CURRENT* view and the default, “over-the-shoulder” lighting, use the following call:

```
(c:scene "N" "DULL" T nil)
```

To create a new scene named SPECIAL using the named view MY_VIEW and the SUN, LAMP, and SPOT lights, issue the following function call:

```
(c:scene "N" "SPECIAL" "MY_VIEW" '("SUN" "LAMP "SPOT"))
```

R—Rename Scene

Renames a scene.

(c:scene "R" *old_name new_name*)

Arguments

old_name	A string that specifies the name of the original scene.
new_name	A string that specifies the new name for the scene.

Examples

Rename a scene from “SPECIAL” to “BRIGHT”:

```
(c:scene "R" "SPECIAL" "BRIGHT")
```

S—Set Scene

Sets the current scene.

(c:scene "S" *[name]*)

Arguments

old_name	A string that specifies the name of the scene to make current.

Return Values

If you omit the *name* argument, **c:scene** returns the name of the currently selected scene.

Examples

Obtain the name of the currently selected scene:

```
Command: (c:scene "S")
"PLAN"
```

If there is no current scene, **c:scene** returns an empty string ("").

To make SCENE3 the current scene, issue the following function call:

```
(c:scene "S" "SCENE3")
```

setuv

Assigns material mapping coordinates to selected objects. Its function has two modes, specified by a string argument (Externally-defined: *render* ARX application)

`(c:setuv` *`mode options`*`)`

The SETUV command lets you assign material mapping coordinates to selected objects.

Arguments

mode	Mode can be one of the following strings:
	A Assign UV mapping to the selection set
	D Detach UV mapping from the selection set
options	Allowable options depend on the *mode* specified.

A—Assign

The "A" (assign) mode assigns mapping coordinates.

Arguments

Arguments expected by this mode depend on whether you specify projection or solid mapping. The assign arguments for projection mapping are described in the following table:

SETUV—"A" mode arguments for projection mapping

Argument	Data type	Description	Default
ssname	PICKSET	The selection set that contains the entities to which you want to assign mapping coordinates	None
mapping type	STR	Type of projection mapping: P—planar D—cylindrical F—spherical	None

SETUV—"A" mode arguments for projection mapping (*continued*)

Argument	Data type	Description	Default
pt1, pt2, pt3	LIST	Three points that define the mapping geometry: Planar—lower-left corner, lower-right corner, upper-left corner Cylindrical—center bottom, center top, direction toward the seam Spherical—center of the sphere, radius (north), direction toward the seam	None
rep	INT	Tiling: 0—no tiling (crop) 1—tile (repeat pattern)	1
scale	LIST (of reals)	The U and V scale factors	(1.0 1.0)
offset	LIST (of reals)	The U and V offsets	(0.0 0.0)

For solid mapping, the option arguments specify only the mapping points. These implicitly define the scale in the UVW dimensions. The assign arguments for solid mapping are described in the following table:

SETUV—"A" mode arguments for solid mapping

Argument	Data type	Description	Default
ssname	PICKSET	The selection set that contains the objects to which you want to assign mapping coordinates	None
mapping type	STR	R—solid	None
pt1	LIST	Point to define the origin	None
pt1	LIST	Point to define the U axis	None
pt1	LIST	Point to define the V axis	None
pt1	LIST	Point to define the W axis	None

Examples

The following function call assigns cylindrical mapping coordinates to an object the user chooses, using tiling and the default scale and offset:

```
(c:setuv "A" (ssget) "C" '(5.0 5.0 5.0) '(5.0 5.0 10.0)
'(10.0 0.0 0.0) 1)
```

D—Detach

The “D” (detach) mode detaches the UV mapping assigned to the objects in the selection set. These objects will now be mapped with the default mapping coordinates until you assign mapping coordinates again.

Arguments

ssname	The selection set that contains the objects from which you want to detach mapping coordinates

Examples

The following call prompts the user for entities that will be detached from their mapping coordinates:

```
(c:setuv "D" (ssget))
```

showmat

Lists the material type and attachment method for a selected object (Externally-defined: *render* ARX application)

`(c:showmat `*`arg1`*`)`

This function lists the material type and attachment method based on *`arg1`*.

Arguments

arg1	Can be an entity name, an integer representing an ACI value, or a layer name (a string).

solprof

Creates profile images of three-dimensional solids (Externally-defined: *solids* ARX application

```
(c:solprof args ...)
```

Arguments

args	The order, number, and type of arguments are the same as those specified when issuing SOLPROF at the Command prompt.

stats

Displays statistics for the last rendering (Externally-defined: *render* ARX application)

```
(c:stats [filename |nil])
```

The STATS command provides information about your last rendering.

Arguments

filename | `nil`	A string specifying the name of the file to save the rendering information in, or `nil` to tell RENDER to stop saving statistics. If you omit the file name, **`c:stats`** displays the Statistics dialog box.

Examples

The following command writes statistics from your last rendering to the *figures.txt* file:

```
(c:stats "figures.txt")
```

If the file already exists, the statistics are appended.

The following command saves the information associated with the last rendering to the *stats.txt* file, and also saves the information associated with the following renderings to this file:

```
(c:stats "stats.txt")
```

The following command tells RENDER to stop saving statistics:

```
(c:stats nil)
```

Index