Introduction

AutoLISP®, an implementation of the LISP programming language, is an integral part of the AutoCAD® package. AutoLISP lets users and AutoCAD developers write macro programs and functions in a powerful high-level language that is well suited to graphics applications. AutoLISP is easy to learn and use, and is very flexible.

Numerous AutoLISP programs, including many of the examples in this manual, are supplied with AutoCAD in the *sample/* directory and on the bonus CD-ROM. Many are also available as shareware and from other third-party developers. Learning how to access and load AutoLISP files will increase your productivity and enhance AutoCAD's flexibility. You may decide, after using some of these programs, that you would like to create your own applications; since AutoLISP code can also be entered at the command line you will find it very easy to learn and experiment with. You may also find that once you start learning AutoLISP, you will use it to augment AutoCAD's commands.

If you are not interested in writing AutoLISP programs you should refer to the discussion of AutoLISP in the *AutoCAD Customization Manual*. This will provide you with a basic understanding of AutoLISP and how to load the programs you acquire.

This is a reference manual; it is not a LISP programming tutorial. Although practical examples in the use of AutoLISP are provided, we recommend that you obtain various texts on LISP in order to learn the programming language. Suggestions are *LISP* by Winston and Horn (second edition) and *Looking at LISP* by Tony Hasemer, both published by Addison-Wesley. LISP is a language that has many dialects, including MacLISP, InterLISP, ZetaLISP, and Common LISP. Many other texts on LISP and AutoLISP are available in your local computer bookstore. AutoLISP adheres most closely to the syntax and conventions of Common LISP, but AutoLISP is a small subset and has many additional functions specific to AutoCAD. This reference manual lists all of the AutoLISP functions and explains how you use them.

AutoLISP was based on XLISP, a program developed by David Betz. We gratefully acknowledge the contribution his work made to the development of AutoLISP.

Why LISP?

We chose LISP as the first AutoCAD application interface language for several reasons:

- LISP is among the easiest of all programming languages to learn and to master.
- It is the chosen language for research and development of artificial intelligence and expert systems.
- Because of LISP's simple syntax, a LISP interpreter is fairly easy to implement and requires little memory.
- A LISP interpreter is ideally suited to the unstructured interaction that characterizes the design process.
- LISP excels at working with collections of heterogeneous objects in various sized groups, which is precisely the type of information a CAD system like AutoCAD manipulates.

The introduction of the AutoCAD Development System™ (ADS®) in Release 11 has provided an additional language for the development of applications. Autodesk is committed to long-term support for AutoLISP. ADS is offered as an alternative.

Organization of the Manual

This manual is a reference for AutoLISP programmers; it also includes some descriptive sections and code examples to supplement the function definitions. It is organized as follows:

- This introduction provides preliminary information on AutoLISP and how to use this manual. It also briefly describes the new and revised AutoLISP functions.
- Chapter 1 contains an introduction to programming in AutoLISP and describes the methods for defining and automatically loading functions.
- Chapters 2 and 3 provide an in-depth look at some of the AutoLISP functions. These chapters include several programming examples.
- Chapter 4 contains a full description of the AutoLISP functions, organized alphabetically.
- Chapter 5 describes methods for memory management and programming techniques. Some of these techniques are used in the sample routines provided with AutoCAD. You should also see the *AutoCAD Extras Manual* for information on other applications.
- Appendix A lists the set of AutoLISP and ADS functions, for comparison and as a quick reference to arguments.
- Appendix B is a reference to Drawing Interchange Format (DXF™) group codes. Its tables are organized both by number and by entity type.
- Appendix C lists the symbolic values of error codes that are reported by the AutoCAD system variable ERRNO.

- Appendix D explains the error messages returned by AutoLISP.
- Appendix E is an AutoLISP tutorial. It walks you through the creation of a new AutoCAD command that uses a dialogue box for user input.
- Appendix F is a decimal and octal ASCII codes chart.

Notational Conventions

This reference manual uses certain conventions to describe the behaviour of functions. For example:

```
(moo string number ...)
```

The function name is shown as you should enter it. The italicized items following the function name indicate the number and type of arguments expected by the function.

In this example, the function `moo` has two required arguments: a string and a number. The ellipsis (". . .") indicates that additional numeric arguments may be supplied to the function. Do not include the ellipsis when you invoke the function. Given the format of a `moo` function call just shown, the following are valid calls to the `moo` function:

```
(moo "Hello" 5)
(moo "Hi" 1 2 3)
```

The following examples do not adhere to the prescribed format and result in errors:

`(moo 1 2 3)`	*First argument must be a string*
`(moo "Hello")`	*Must have at least one numeric argument*
`(moo "do" '(1 2))`	*Second argument must be a number, not a list*

Optional arguments are shown enclosed in square brackets ("[]") and unless followed by an ellipsis may be used only once, as in:

```
(foo string [number])
```

Here, the function `foo` requires one string argument and accepts one optional numeric argument. For example, the following are valid calls to the `foo` function:

```
(foo "catch")
(foo "catch" 22)
```

The following examples do not adhere to the prescribed format and result in errors:

`(foo 44 13)`	*First argument must be a string*
`(foo "foe" 44 13)`	*Too many arguments*

Typeface Conventions

This manual uses the following typeface conventions:

Convention	Use
Serif typeface	
Initial Caps	Names of drawing entities: Arcs, Circles, Lines
UPPERCASE	Names of AutoCAD commands: SAVE, COPY, STATUS AutoCAD system variables, their values, and environment variables: CMDECHO, DIMZIN, CONTINUOUS, ACAD, ACADXMEM
Italic	Filenames, pathnames, and filename extensions: *acad.lsp, /usr/acad/acad, .mnu*
Sans serif typeface	
Boldface	Names of operating system (e.g., MS-DOS® or UNIX®) commands: At the DOS prompt, enter **dir**. At the UNIX prompt, enter **ls**. Text that the user enters. You would enter the PLINE command at the AutoCAD Command: prompt: Command: **pline**
Not bold	Prompts and other text displayed on screen: Command: Result is 10.51
`Courier typeface`	
`Boldface`	AutoLISP and C function names: **`command, ads_command()`**
`Not bold`	AutoLISP and C variables, types, values, and sample listings of AutoLISP and C code: `pt1, ads_point, NULL`
`Italic`	Formal arguments specified in function definitions: `(distof` *`string [mode]`*`)` The **`distof`** function requires one argument, *`string`*, and optionally accepts *`mode`*.
Key caps	Keys on the keyboard: Delete , ScrollLock , Ins , Shift , etc. The Enter or Return key appears like this: Return or ↵ When you must press two keys simultaneously, a plus sign connects the two keys: Ctrl+C

Recent Changes and Enhancements

Areas of major changes include the following:

- The PLOT command is now accessible from AutoLISP.
- AME commands can be called from an AutoLISP routine; see the *AME Reference Manual* for a complete description.
- ASE commands can also be called from AutoLISP routines; see the *ASE Reference Manual*.
- The **ssget** function now supports new entity selection methods in addition to various combinations of entity filtering of group codes, relational tests, and Boolean functions.
- The **entsel** and **nentsel** functions now return keywords as well as pick points.
- The new **nentselp** function is similar to **nentsel** but allows point specification without user input.
- The **initget** function now lets user-input functions accept arbitrary keyboard input.
- The new functions **angtof** and **distof** let you convert strings representing angles and distances to reals.
- The new **textbox** function lets you retrieve text extents.
- The new **tablet** function provides control over digitizer calibration.
- The new **grvecs** function lets you display multiple vectors on the graphics screen.
- The new **alert** function displays an alert box with a warning message supplied by the application.
- The new **getfiled** function lets an AutoLISP application prompt the user for a filename by displaying the standard AutoCAD file dialogue box.
- The atomlist symbol has been replaced by a new **atoms-family** function.

As installed, the *acad/* directory contains a file named *readme.doc*. Read this file; it describes last-minute changes and updates to the AutoCAD and AutoLISP documentation.

Programming Interfaces to Standard External Applications

Some external (ADS) applications that are now standard to AutoCAD provide features that your applications can access. These functions are defined when the file *acadapp* (this file has a *.exp* extension on DOS platforms) is loaded. They fall into two categories:

- Externally defined AutoCAD functions

  The functions **acad_helpdlg** and **acad_colordlg** allow you to display the standard AutoCAD Help and colour selection dialogue boxes. The function **acad_strlsort** sorts a list of strings alphabetically.

- Application programming interfaces to interactive AutoCAD commands

  Commands that AutoCAD users would normally access interactively, using dialogue boxes, can be invoked as external functions (using the `c:xxx` form). These commands include BHATCH, BPOLY, PSDRAG, PSIN, and PSFILL.

New sections in chapter 4 describe these functions and how to use them. See “ADS Defined AutoLISP Functions” on page 172 and “ADS Defined Commands” on page 174.

Programmable Dialogue Boxes

Starting with Release 12 of AutoCAD, you can design and implement your own dialogue boxes, similar to those AutoCAD uses. Support for dialogue boxes is independent of the hardware and operating system you are using. The behaviour of a dialogue box remains essentially the same across all platforms, while its appearance changes to that of the platform’s *graphical user interface (GUI)*.

The design of dialogue boxes is defined in ASCII text files written in *Dialogue Control Language (DCL)*. The DCL description of a box determines what it contains: buttons, lists, text, and so on, and how these components relate to each other. The use and behaviour of a dialogue depends on the application that employs it. Both AutoLISP and the AutoCAD Development System (ADS) provide functions for handling dialogue boxes.

To learn more about creating your own dialogue boxes, see chapter 9, “Programmable Dialogue Boxes,” in the *AutoCAD Customization Manual.*

Chapter 1
Essentials of Using AutoLISP

Data Types in AutoLISP

AutoLISP supports the following data types:

- Symbols
- Lists
- Strings
- Integers
- Real numbers
- File descriptors
- AutoCAD entity names
- AutoCAD selection sets
- Subrs (built-in functions)
- External Subrs (ADS functions)

Whenever AutoCAD asks you for input of a certain type (a point or a scale factor, for example), you can use an AutoLISP expression of that type, or an AutoLISP function that returns that type of result, to supply the desired value.

Symbols

AutoLISP uses symbols to store values. The following code example uses the **setq** function to set the symbol `pt1` to the point value (1,2):

```
(setq pt1 '(1 2))
```

The terms "symbol" and "variable" are used interchangeably.

Lists

AutoLISP uses lists extensively. They provide an efficient method of storing numerous, related values in one symbol. Several AutoLISP functions provide a basis for programming two-dimensional and three-dimensional graphics applications; these functions return point values in the form of a list.

To deal with graphics coordinates, AutoLISP observes the following conventions:

2D points are expressed as lists of two real numbers (*X Y*), as in

```
(3.4 7.52)
```

The first value is the *X* coordinate and the second value is the *Y* coordinate.

3D points are expressed as lists of three real numbers (*X Y Z*), as in

```
(3.4 7.52 1.0)
```

The first value is the *X* coordinate, the second value is the *Y* coordinate, and the third value is the *Z* coordinate.

Strings

Strings can be of any length; memory for them is dynamically allocated. Although string constants are limited to 132 characters, strings of unlimited length can be created by using the **`strcat`** function to join strings together.

Integers

Integers are whole numbers entered without a decimal point. AutoLISP integers are 32-bit signed numbers with values between –2,147,483,648 and +2,147,483,647. Although AutoLISP uses 32-bit values internally, those transferred between AutoLISP and AutoCAD are restricted to 16-bit values (i.e., you cannot pass a value greater than +32767 or less than –32768 to AutoCAD). If you are using a value that exceeds these limits, you can use the **`float`** function to convert it to a real, since reals are passed as 32-bit values.

Reals

A real is a number containing a decimal point. Numbers between –1 and 1 must contain a leading zero. Real numbers are stored in double-precision floating-point format, providing at least 14 significant digits of precision, even though the AutoCAD command line area shows only 6 significant digits.

File Descriptors

File descriptors are alphanumeric labels assigned to files opened by AutoLISP. When an AutoLISP function needs to access a file (for reading or writing), its label must be referenced. The example below opens the file *myinfo.dat*, making it accessible to other functions for reading, and assigns the value of the file descriptor to the symbol `fil`:

`(setq fil (open "myinfo.dat" "r"))` might return <File: #34614>

Entity Names

An entity name is a numeric label assigned to entities in a drawing. It is actually a pointer into a file maintained by AutoCAD, from which AutoLISP can find the entity's database record and its vectors (if on screen). This label can be referenced by AutoLISP functions to allow selection of entities for processing in various ways. The following example sets the symbol `e1` to the entity name of the last entity entered into the drawing:

`(setq e1 (entlast))` might return <Entity name: 60000016>

Selection Sets

Selection sets are groups of one or more entities. As with the regular entity selection process in AutoCAD, you can interactively add objects to, or remove objects from, selection sets with AutoLISP routines. The following example assigns the selection set consisting of the previously selected objects to the symbol `ssprev`:

`(setq ssprev (ssget "P"))` might return <Selection set: 1>

Subrs and External Subrs

All AutoLISP functions described in this manual are *Subrs*, which are built-in subroutines. A Subr can be redefined with the **`defun`** function; however, its original use will no longer be available to other routines (redefinition of a Subr is *not* recommended). An *External Subr* is a subroutine defined by an ADS application.

Lexical Conventions

AutoLISP input can take several forms. It can be entered from the keyboard at the AutoCAD prompt line, read from an ASCII file, or read from a string variable. In all cases, these conventions must be followed:

- Symbol names can consist of any sequence of printable characters except:

  `( ) . ' " ;`

- The following characters terminate a symbol name or numeric constant:

  ```
  ( ) ' " ; (space) (end of line)
  ```

- Expressions can span multiple lines.
- Multiple spaces between symbols are equivalent to a single space. Although you don't have to indent the lines of your AutoLISP programs, doing so makes the structure of your functions more obvious. Using tabs in AutoLISP is discouraged; typically they are read as spaces, but some platforms might interpret them differently.
- Symbol and function names (Subrs) are not case-sensitive in AutoLISP; they can be entered uppercase or lowercase.
- Integer constants can begin with an optional + or – character. As mentioned earlier, their range is –2,147,483,648 to +2,147,483,647.
- Real constants can begin with an optional + or – character and consist of one or more numeric digits, followed by a decimal point, followed by one or more numeric digits (i.e., .4 is not recognized as a real; 0.4 is correct). Similarly, 5. is not recognized as a real; 5.0 is correct. Reals can be expressed in scientific notation, which has an optional e or E followed by the exponent of the number (i.e., 0.0000041 is the same as 4.1e-6).
- Literal strings are sequences of characters surrounded by double quotes. Within quoted strings the backslash (\) character allows control characters (or escape codes) to be included.

  These are the codes currently recognized:

  Table 1–1. Escape codes

Code	Meaning
`\\`	\ character
`\"`	" character
`\e`	Escape character
`\n`	Newline character
`\r`	Return character
`\t`	Tab character
`\`*nnn*	Character whose octal code is *nnn*

  For instance, the following issues a prompt on a new line:

  ```
  (prompt "\nEnter first point: ")
  ```

  and

  ```
  (princ "\361")
  ```

  would display the " ± " symbol on DOS systems.

- The single quote character can be used as shorthand for the `quote` function. Thus:

  `'foo` is equivalent to `(quote foo)`

- Comments can be included in AutoLISP program files. Comments begin with a semicolon and continue through the end of the line. For example:

  ```
  ; This entire line is a comment
  (setq area (* pi r r)) ; Compute area of circle.
  ```

  Any text within " ;| ... |; " is ignored; this allows comments to be included within a line of code or extend for multiple lines. Following is an example of an *inline* comment:

  ```
  (setq tmode ;|some note here|; (getvar "tilemode"))
  ```

  and this shows a comment that continues for multiple lines:

  ```
  (setvar "orthomode" 1) ;|comment starts here
  and continues to this line,
  but ends way down here|; (princ "\nORTHOMODE set On.")
  ```

The AutoLISP Evaluator

At the core of every LISP interpreter is the evaluator. The evaluator takes a line of user input, evaluates it, and returns a result. The following is the process of evaluation in AutoLISP:

- Integers, reals, strings, file pointers, and Subrs evaluate to themselves.
- Symbols evaluate to the value of their current binding.
- Lists are evaluated according to the first element of the list.
  - If the first element of a list evaluates to a list (including `nil`), the list is assumed to be a function definition and the function is evaluated using the values of the remaining list elements as arguments.
  - If the first element of a list evaluates to the name of an internal function (Subr), the remaining list elements are passed to the Subr as the formal arguments and are evaluated by the Subr.

AutoLISP Expressions

All AutoLISP expressions have this form:

(function-name [arguments] . . .)

Each expression begins with a left parenthesis and consists of a function name and an optional list of arguments to that function (each of which can itself be an expression). The expression then ends with a right parenthesis. Every expression returns a value that can be used by a surrounding expression; if there is no surrounding expression, AutoLISP returns the value to AutoCAD.

If you enter an AutoLISP expression in response to the AutoCAD Command: prompt, AutoLISP evaluates the expression and prints the result. The AutoCAD Command: prompt then reappears. When printing real numbers, AutoLISP displays up to 6 significant digits.

If an incorrect expression is entered or read from a file, AutoLISP might display the following prompt:

n>

where *n* is an integer indicating how many levels of left parentheses remain unclosed. If this prompt appears, you must enter *n* right parentheses to exit from this condition. A common mistake is to omit a closing double quote (") in a text string, in which case the right parentheses are interpreted as being quoted and have no effect in changing *n*. To correct this condition, cancel the function by entering Ctrl+C and reenter it correctly.

AutoLISP Variables

AutoLISP variables can be of four types: integer, real, point, and string. A variable's type is automatically attached to it based on the type of value assigned. Variables retain their values until reassigned or until the current drawing session has ended. You can name your variables anything you want, provided the first character is alphabetic. The variable `pi` is preset to the value of π. You can use it just like variables you define yourself.

You use the AutoLISP **setq** function to assign values to variables. This is the format:

```
(setq variable-name value)
```

The **setq** function assigns the specified value to the variable whose name is given. It also returns the value as its function result. If you use **setq** when AutoCAD has issued a Command: prompt, it will set the variable and display the value assigned. Note the parentheses surrounding this expression; they are required. A few examples follow.

```
(setq k 3)
(setq x 3.875)
(setq layname "EXTERIOR-WALLS")
```

These expressions assign values to an integer, a real, and a string variable, respectively. Point variables are more complicated, since they contain *X*, *Y*, and (optionally) *Z* components. Points are expressed as *lists* of two or three numbers surrounded by parentheses, as in the following:

```
(3.875 1.23)          a 2D point
(88.0 14.77 3.14)     a 3D point
```

The first item in the list is the *X* component of the point; the second is the *Y* component; and the third (if present) is the *Z* component. You can use another built-in function **list** to form such lists.

```
(list 3.875 1.23)
(list 88.0 14.77 3.14)
```

Thus, to assign particular coordinates to a point variable, you can use one of the following expressions:

```
(setq pt (list 3.875 1.23))
(setq pt (list 88.0 14.77 3.14))
(setq pt (list abc 1.23))
```

The latter uses the value of variable `abc` as the *X* component of the point.

You can refer to *X*, *Y*, and *Z* components of a point individually, using three more built-in functions called **car**, **cadr**, and **caddr**.

`(car pt)`	*Returns the X component of point variable* `pt`
`(cadr pt)`	*Returns the Y component of point variable* `pt`
`(caddr pt)`	*Returns the Z component of point variable* `pt`

For example, suppose that variables `pt1` and `pt2` are set to points (1.0,2.0) and (3.0,4.0), defining the lower-left and upper-right corners of a rectangle. We can use the **car** and **cadr** functions to set variable `pt3` to the upper-left corner of the rectangle, by extracting the *X* component of `pt1` and the *Y* component of `pt2` as follows:

```
(setq pt3 (list (car pt1) (cadr pt2)))
```

The above expression would set `pt3` equal to point (1.0,4.0).

If you want to use the value of a variable as the response to a prompt from AutoCAD, simply enter the name of the variable, preceded by an exclamation point, !. Suppose, for example, that you've set variable `abc` to the value 14.88702. You could then enter **!abc** any time you want to respond to a prompt with the value 14.88702. For instance:

```
Column distance: !abc
```

Similarly, if you want to begin drawing a line at point (1.0,4.0) and you've set variable `pt` equal to that point, you can enter this:

```
Command: line
From point: !pt
. . .
```

Notes:

1. In order for expressions or variable references to be interpreted correctly, the left parenthesis, (, or exclamation point, !, must be the *first character* you enter in response to a prompt.
2. Since ordinary text strings can begin with ! or (, you must set the system variable TEXTEVAL to 1 if you want to use a variable or expression to supply the text string to such commands as TEXT and ATTDEF.
3. You cannot use a variable reference to issue an AutoCAD command. For example, if you set variable `x` to the string `"line"` and then enter **!x** in response to the AutoCAD Command: prompt, AutoCAD will simply display the value `"line"`; the LINE command will *not* be executed. The **command** function (discussed on page 98) executes AutoCAD commands from within AutoLISP functions.

Defining Functions and Automatic Loading

You can save function definitions in files with an extension of *.lsp* and load them, using the AutoLISP **load** function, described on page 133, or include them in an *acad.lsp* file to be loaded automatically each time AutoCAD is started. Loading a *.lsp* file causes evaluation of its expressions. Most commonly a *.lsp* file uses the **defun** function to store groups of functions in the computer's memory for later execution (see the **defun** function on page 101).

If a function is defined with a name of the form C:*XXX*, it can be issued at the AutoCAD prompt line in the same manner as a built-in AutoCAD command. The following section describes this concept.

Important: You should consider the **S::** function name prefix to be *reserved.* To avoid conflicts with unrelated functions, use this prefix only for the special function **S::STARTUP** described on page 16.

C:XXX Functions—Adding Commands to AutoCAD

You can add new commands to AutoCAD by using **defun** to define functions implementing those commands. To be used as AutoCAD commands, such functions must adhere to the following rules:

1. The function must have a name of the form **C:*XXX*** (upper- or lowercase characters). The **C:** portion of the name must always be present; the ***XXX*** portion can be a command name of your choice. **C:*XXX*** functions can be used to override built-in AutoCAD commands if those commands have been undefined with the UNDEFINE command (see the UNDEFINE command in the *AutoCAD Reference Manual*).

   Note: In this case, **C:** is a special prefix that denotes a command line function; it is not a reference to a disk drive.

2. The function must be defined with a nil argument list (local symbols are permitted; local symbols are discussed with the **setq** function on page 150).

Example

The following function uses a Polyline to draw a square.

```
(defun C:PSQUARE (/ pt1 pt2 pt3 pt4 len)
    (setq pt1 (getpoint "Lower left corner: "))
    (setq len (getdist pt1 "Length of one side: "))
    (setq pt2 (polar pt1 0.0 len))
    (setq pt3 (polar pt2 (/ pi 2.0) len))
    (setq pt4 (polar pt3 pi len))
    (command "pline" pt1 pt2 pt3 pt4 "C")
)
```

Once loaded (with the **load** function) functions defined like this can be invoked by simply entering the ***XXX*** part of the function name at the AutoCAD prompt line. If ***XXX*** is not a known command, AutoCAD tries calling the

AutoLISP function **C:XXX** with no parameters. For the sample **C:PSQUARE** function, the command sequence looks like this:

```
Command: psquare
Lower left corner: Select a point.
Length of one side: Enter a distance.
```

The function then invokes the AutoCAD PLINE command and responds to its prompts to draw the requested square.

Adding commands to AutoCAD in this manner is a very powerful feature of AutoLISP. Once defined, the new command can use all the facilities afforded by AutoLISP. Actual use of the new command does not require you to surround the command name with parentheses, so this AutoLISP-implemented command is used just like any other AutoCAD command.

A function defined in this manner can be issued transparently from within any prompt of any built-in AutoCAD command, provided that the function issued transparently does not call the **command** function. When issuing a **C:*XXX*** defined command transparently, you must precede the ***XXX*** portion with a " ' " (i.e., **'PSQUARE**). If you want to issue a transparent command while a **C:*XXX*** command is active, you must precede it with a " ' " (as with all commands issued transparently).

When calling a function defined as a command from the code of another AutoLISP function, you must use the whole name (i.e., **(C:PSQUARE)**).

Note: You typically can't respond to prompts from an AutoLISP-implemented command with an AutoLISP statement. However, if your AutoLISP routine makes use of the **initget** function, arbitrary keyboard input is permitted with certain functions; this can let an AutoLISP-implemented command accept an AutoLISP statement as response. Also, the values returned by a DIESEL expression can perform some evaluation of the current drawing and return these values to AutoLISP. See chapter 8 of the *AutoCAD Customization Manual* for information on the DIESEL String Expression Language.

Function Libraries—Automatic Loading

As you create a *library* of useful AutoLISP routines, you may want them to be automatically loaded each time you start AutoCAD. If the *acad.lsp* file exists in the AutoCAD library path, it is loaded automatically when you start AutoCAD and each time you start a new drawing.

Another type of AutoLISP file that can be automatically loaded is a *.mnl* file. These files typically contain AutoLISP routines required for the proper operation of a menu file (*.mnu*). When a menu file is loaded (either by starting a drawing or issuing the MENU command), AutoCAD searches the directory containing the newly loaded *.mnu* file for a *.mnl* file of the same filename. If a matching *.mnl* file is found AutoCAD loads the AutoLISP code in that file after loading the menu file (e.g., AutoCAD loads the file *acad.mnl* after loading the *acad.mnx* compiled menu file).

If a menu file is loaded with the AutoLISP (command) function, its associated *.mnl* file is not loaded until the entire AutoLISP routine has run to completion.

The *acad.lsp* and *.mnl* files are not required. If either or both exist, AutoCAD loads the *acad.lsp* file first, then the associated *.mnl* file.

The AutoCAD library path is searched in the following order:

1. The current directory
2. The directory containing the current drawing file
3. The directories named by the ACAD environment variable (if this variable has been specified)
4. The directory containing the AutoCAD program files

Your *acad.lsp* and *.mnl* files can define the desired AutoLISP functions directly, or they can use the **load** function to load them from other files. The latter method makes editing these files easier and reduces their size.

Following is an example of the possible contents of an *acad.lsp* or *.mnl* file (the files requested by the following **load** function calls are assumed to be in the library path):

```
(load "3darray")      loads the file 3darray.lsp
(load "chgtext")      loads the file chgtext.lsp
(load "setup")        loads the file setup.lsp
(load "new_func")     loads the file new_func.lsp
(princ)               exits the file quietly
```

You can use these features to create a *library* of useful functions and ensure that they are always present when you need them.

Note: You should not use the command function or any other function that accesses the drawing database directly from an *acad.lsp* or *.mnl* file; since the drawing is not fully initialized at this point, unpredictable results can occur. These types of function calls can be included in a **S::STARTUP** function.

To make your *acad.lsp* file execute a series of **command** statements automatically after loading a drawing, include in it a **defun** of the special function **S::STARTUP**.

S::STARTUP Function—Automatic Execution

If the user-defined function **S::STARTUP** is included in the *acad.lsp* or *.mnl* file, it is called automatically (with no arguments) when you enter a new drawing or open an existing drawing. Thus, you can include a **defun** of **S::STARTUP** in your *acad.lsp* file to perform any setup operations you want at the start of an editing session.

Continued on page 17

Example

Suppose you wanted to override the standard AutoCAD QUIT and END commands with versions of your own. You can do so with an *acad.lsp* file containing the following:

```
(defun C:QUIT ()
    ... your definition ...
)
(defun C:END ()
    ... your definition ...
)
(defun S::STARTUP ()
    (command "undefine" "quit")
    (command "undefine" "end")
)
```

Before the drawing has been initialized, your new definitions for QUIT and END are defined with the **defun** function; once the drawing has been fully initialized, the **S::STARTUP** function is called and the standard definitions of QUIT and END are undefined.

Since an **S::STARTUP** function can be defined in many places (an *acad.lsp* file, a *.mnl* file, or any other AutoLISP file loaded from either of these) it is possible to overwrite a previously defined **S::STARTUP** function. The following example shows one method of ensuring your startup function works in harmony with others:

```
(defun mystartup ()
    ... your startup function ...
)

(if s::startup
    (setq s::startup (append s::startup ’((mystartup)) ))
    (defun s::startup () (mystartup) )
)
```

Error Handling

If AutoLISP encounters an error during evaluation, it prints a message in this form

Error: *text*

where ***text*** is a description of the error. If the ***error*** function is defined (non-nil), AutoLISP executes that function (with ***text*** passed as its single argument) instead of printing the message. If ***error*** is not defined or is bound to nil, AutoLISP evaluation stops and displays a traceback of the calling function and its callers up to 100 levels deep.

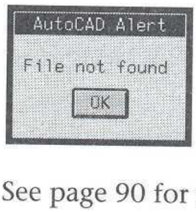
A code for the last error is saved in the AutoCAD system variable ERRNO, where it can be retrieved using the **getvar** function. For more information and examples, see "Error Handling" on page 183. See the error messages in appendix D and the error codes in appendix C. The ***error*** function is described on page 113.

You can also warn the user about error conditions by displaying an *alert box,* a small dialogue box containing a message supplied by your program. To display an alert box, call the **alert** function. Alert boxes are a somewhat more "heavy-handed" way of warning the user, since the user has to push the OK button before continuing.

Example:

The following call to **alert** displays the alert box shown below.

```
(alert "File not found")
```



File not found

See page 90 for more information on the **alert** function.

Chapter 2
General Utility Functions

AutoLISP provides various functions for examining the drawing currently loaded, modifying it, interacting with the AutoCAD user, and so on. This chapter is a general description of these functions, many of which have similar functions in ADS. It introduces the functions, describes how they can be used in conjunction with other functions, and provides code examples of their use. For specific details on calling a particular function, refer to the catalog in chapter 4.

Functions that handle entities, selection sets, and symbol tables are described in chapter 3.

AutoCAD Queries and Commands

The functions described in this section provide direct access to AutoCAD commands and drawing services. Their behaviour depends on the current state of AutoCAD system and environment variables, and on the drawing that is currently loaded.

Command Submission

The most general of the AutoLISP functions that access AutoCAD is **command**. This function sends an AutoCAD command along with other related information directly to the AutoCAD Command: prompt.

The **command** function has a variable-length argument list. These arguments must correspond to the types and values expected by that command's prompt sequence: these may be strings, real values, integers, points, entity names, or selection set names. Data such as angles, distances, and points can be passed either as strings (as the user might enter them) or as the values themselves (as integer or real values, or as point lists). An empty string ("") is equivalent to entering a space or ↵ on the keyboard.

There are some restrictions on the commands that can be invoked with the **command** function; see page 98 for details.

Example

The following code fragment shows a few representative calls to **command**.

```
(command "circle" "0,0" "3,3")
(command "thickness" 1)
(setq p1 '(1.0 1.0 3.0))
(setq rad 4.5)
(command "circle" p1 rad)
```

Provided AutoCAD is at the Command: prompt when these functions are called, AutoCAD performs the following actions:

1. Draws a circle centred at (0.0,0.0) and passes through (3.0,3.0).
2. Changes the current thickness to 1.0.
3. Draws another (extruded) circle centred at (1.0,1.0,3.0) with a radius of 4.5.

Note that the first call to **command** passes points to the CIRCLE command as strings, and the second passes an integer to the THICKNESS command. The last call to **command** uses a 3D point and a real (floating-point) value, both of which are stored as variables and passed by reference to the CIRCLE command.

Pausing for User Input

If an AutoCAD command is in progress and the predefined symbol PAUSE is encountered as an argument to **command**, the command is suspended to allow direct user input (usually point selection or dragging). This is similar to the backslash pause mechanism provided for menus.

If you issue a transparent command while a **command** function is suspended, the **command** function remains suspended. Thus, users can 'ZOOM and 'PAN all they want while at a **command** pause. The pause remains in effect until AutoCAD gets valid input and no transparent command is in progress. For example:

```
(command "circle" "5,5" pause "line" "5,5" "7,5" "")
```

begins the CIRCLE command, sets the centre point at (5,5), and then pauses to let the user drag the circle's radius on screen. When the user picks the desired point (or types in the desired radius), the function resumes, drawing a line from (5,5) to (7,5).

Menu input is not suspended by an AutoLISP pause. If a menu item is active when the **command** function pauses for input, that input request can be satisfied by the menu. If you want the menu item to be suspended as well, you must provide a backslash in the menu item. When valid input is found, both the **command** function and the menu item resume.

Passing Pick Points to AutoCAD Commands

Some AutoCAD commands (such as TRIM, EXTEND, and FILLET) require the user to specify a *pick point* as well as the entity itself. To pass such pairs of entity and point data via the **command** function without the use of a PAUSE, you must first obtain these values and store them as variables. Points can be passed as strings within the **command** function or be defined outside the function and passed as variables, as this example shows.

Example

The following code fragment shows one method of passing an entity name and a pick point to the **command** function.

```
(command "circle" "5,5" "2")              Draws circle
(command "line" "3,5" "7,5" "")           Draws line
(setq el (entlast))                       Gets last entity name
(setq pt '(5 7))                          Sets point pt
(command "trim" el "" pt "")              Performs trim
```

Provided AutoCAD is at the Command: prompt when these functions are called, AutoCAD performs the following actions:

1. Draws a circle centred at (5,5) with a radius of 2.
2. Draws a line from (3,5) to (7,5).
3. Creates a variable el that is the name of the last entity added to the database. See chapter 3 for more discussion on entities and entity functions.
4. Creates a variable pt that is a point on the circle (this point selects the portion of the circle to be trimmed).
5. Performs the TRIM command by selecting the entity el, and by selecting the point specified by pt.

System and Environment Variables

A pair of functions, **getvar** and **setvar**, let AutoLISP applications inspect and change the value of AutoCAD system variables. These functions use a string to specify the variable name (in either uppercase or lowercase). The **setvar** function specifies a value of the type that the system variable expects. It is important to remember that AutoCAD system variables come in various types: integers, real values, strings, 2D points, and 3D points. Values supplied as arguments to **setvar** must be of the expected type; if an invalid type is supplied, an AutoLISP error is generated. See appendix A of the *AutoCAD Reference Manual* for a list of the system variables and their types.

Example

The following code fragment ensures that subsequent FILLET commands will use a radius of at least one:

```
(if (< (getvar "filletrad") 1)
   (setvar "filletrad" 1)
)
```

An additional function **getenv** provides AutoLISP routines access to the currently defined operating system environment variables.

File Search

The **findfile** function lets an application search for a file of a particular name. The application can specify the directory to search, or it can use the current AutoCAD library path.

Example

In the following code fragment, **findfile** searches for the requested filename according to the AutoCAD library path:

```
(setq refname "refc.dwg")
(setq fil (findfile refname))
(if fil
   (setq refname fil)
   (princ (strcat "\nCould not find file " refname ". " ))
)
```

If the call to **findfile** is successful, the variable `refname` is set to a fully qualified pathname string, such as:

```
"/home/work/ref/refc.dwg"
```

Note: When specifying a DOS pathname, you must precede the backslash " \ "with a backslash (" \\ ") to be recognized by AutoLISP (see "Lexical Conventions" on page 9). Alternatively you can use the slash character " / " as a directory separator.

The **getfiled** function displays a dialogue box containing a list of available files of a specified extension type in the specified directory. This gives AutoLISP routines access to the AutoCAD Get File dialogue, promoting a uniform appearance with other AutoCAD commands using this feature.

A call to **getfiled** takes four arguments which determine the appearance and functionality of the dialogue box. The application must specify the following string values, each of which can be `nil` if desired: a title, which is placed at the top of the dialogue; a default filename, displayed in the edit box at the bottom of the dialogue; and an extension type, which determines the initial files provided for selection in the list box. The final argument is an integer value that specifies how the dialogue interacts with selected files.

Example

This simple routine uses **getfiled** to let you view your directory structure and select a file:

```
(defun C:DDIR()
   (setq dfil (getfiled "Directory Listing" "" "" 2))
   (princ (strcat "\nVariable 'dfil' set to selected file " dfil "."))
   (princ)
)
```

This can be a very useful utility command. The variable `dfil` is set to the file you select, which can then be used by other AutoLISP functions or as a response to a command line prompt for a filename; to use this variable in response to a command line prompt you would enter **!dfil**.

Note: You can't use **!dfil** in a dialogue box; it is valid only at the command line.

See also: Page 118 for a detailed explanation of the **getfiled** function.

Object Snap

The **osnap** function finds a point via one of the AutoCAD Object Snap modes. The snap modes are specified in a string argument.

Examples

The following call to **osnap** looks for the midpoint of an entity near `pt1`:

```
    (setq pt2 (osnap pt1 "midp"))
```

The following call looks for either the midpoint, endpoint, or centre of an entity nearest `pt1`:

```
    (setq pt2 (osnap pt1 "midp,endp,center"))
```

In both examples, `pt2` is set to the snap point if one is found that fulfills the osnap requirements. Otherwise `pt2` is set to `nil`.

Important: The system variable APERTURE determines the allowable proximity of a selected point to an entity when using Object Snap.

Geometric Utilities

A group of functions allow applications to obtain geometric information. The **distance** function finds the distance between two points, **angle** finds the angle in radians between a line and the *X* axis (of the current UCS), and **polar** finds a point via polar coordinates (relative to an initial point). The **inters** function finds the intersection of two lines.

Note: Unlike **osnap**, the functions in this group simply calculate the point, line, or angle values, and do not actually query the current drawing.

Examples

The following code fragment shows some simple calls to the geometric utility functions:

```
(setq pt1 ’(3.0 6.0 0.0))
(setq pt2 ’(5.0 2.0 0.0))
(setq base ’(1.0 7.0 0.0))
(setq rads (angle pt1 pt2))             Angle in XY plane of current UCS
                                        (value is returned in radians)
(setq len (distance pt1 pt2))           Distance in 3D space

(setq endpt (polar base rads len))
```

The call to **polar** sets `endpt` to a point that is the same distance from (1,7) as `pt1` is from `pt2`, and at the same angle from the *X* axis as the angle between `pt1` and `pt2`.

Text Box Utility Function

The `textbox` function returns the diagonal coordinates of a box that encloses a Text entity. It takes a list of the type returned by `entget` (an association list of group codes and values) as its single argument. This list can contain a complete association list description of the Text entity or just a list describing the text string. Entity definitions and association lists are discussed in greater detail in the following chapter.

The points returned by `textbox` describe the bounding box of the Text entity as if its insertion point were located at (0,0,0) and its rotation angle were 0. The first list returned is generally the point (0.0 0.0 0.0) unless the Text entity is oblique, vertical, or 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. The returned point lists always describe the bottom-left and upper-right corners of this bounding box, regardless of the orientation of the Text being measured.

Examples

This example shows the minimum allowable entity list that `textbox` accepts. Since no additional information is provided, `textbox` uses the current defaults for text style and height.

```
Command: (textbox '((1 . "Hello world")) ) ↵
((0.0 0.0 0.0) (2.80952 1.0 0.0))
```

This example demonstrates one method of providing the `textbox` function with an entity list.

```
Command: dtext
Justify/Style/<Start point>: 1,1
Height <1.0000>: ↵
Rotation angle <0>: ↵
Text: test
Text: ↵
Command: (setq e (entget (entlast)))
((-1 . <Entity name: 6000001c>) (0 . "TEXT") (8 . "0")
(10 1.0 1.0 0.0) (40 . 1.0) (1 . "test") (50 . 0.0)
(41 . 1.0)(51 . 0.0) (7 . "STANDARD") (71 . 0) (72 . 0)
(11 0.0 0.0 0.0) (210 0.0 0.0 1.0) (73 . 0))
Command: (textbox e)
((0.0 0.0 0.0) (0.8 0.2 0.0))
```

Figure 2-1 shows the results of applying `textbox` to a Text entity with a height of 1.0. The figure also shows the baseline and insertion point of the text.

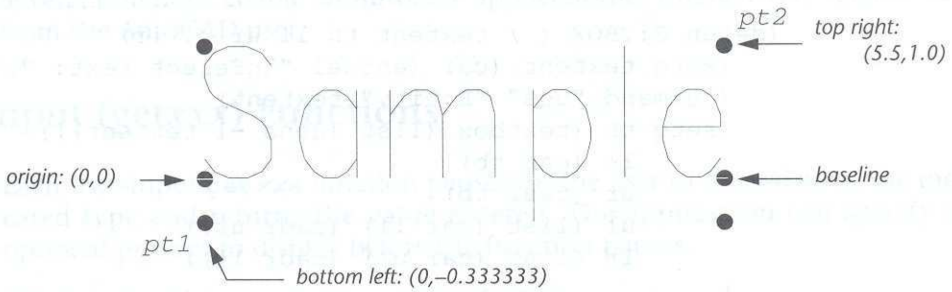


Figure 2–1. Points returned by textbox

If the text is vertical or rotated, `pt1` and `pt2` are still in left-to-right, bottom-to-top order; the bottom-left point might have negative offsets, if necessary.

Figure 2-2 shows the point values (`pt1` and `pt2`) that `textbox` returns for samples of vertical and aligned text. In both samples, the height of the letters is 1.0 (for the aligned text, the height is adjusted to fit the alignment points).

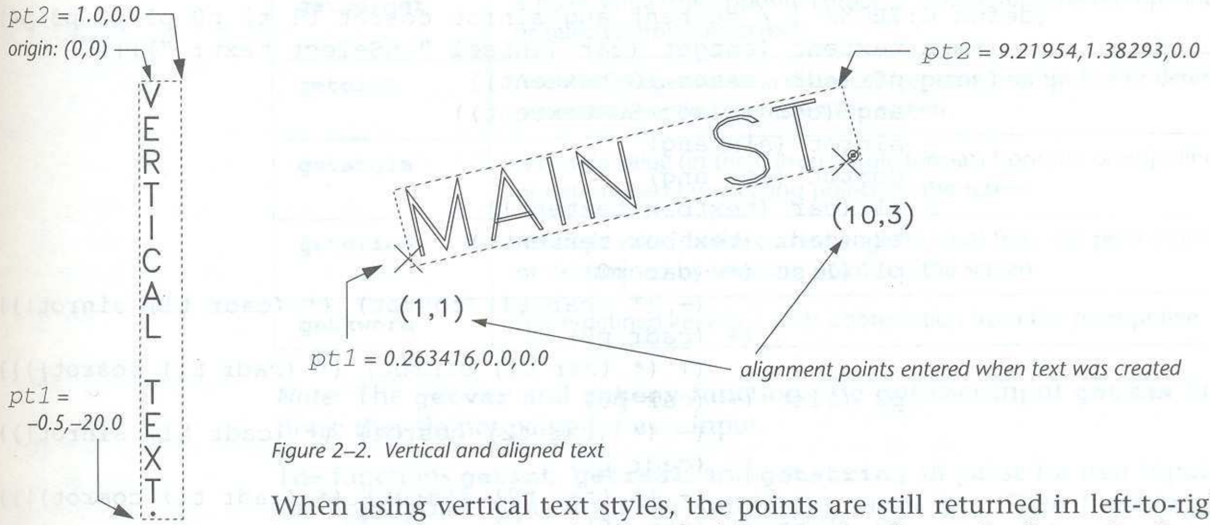


Figure 2–2. Vertical and aligned text

When using vertical text styles, the points are still returned in left-to-right, bottom-to-top order as they are for horizontal styles, so the first point list will contain negative offsets from the text insertion point.

Regardless of the text orientation or style, the points returned by `textbox` are such that the text insertion point (group code 10) directly translates to the origin point of the Entity Coordinate System (ECS). This point can be referenced when translating the coordinates returned from `textbox` into points that define the actual extents of the text.

Following are two sample routines that use `textbox` to place a box around selected text regardless of its orientation.

Examples

This routine uses the **textbox** function to draw a box around a selected Text entity.

```
(defun C:TBOX ( / textent tb ll ur ul lr)
   (setq textent (car (entsel "\nSelect text: ")))
   (command "ucs" "Entity" textent)
   (setq tb (textbox (list (cons -1 textent)))
         ll (car tb)
         ur (cadr tb)
         ul (list (car ll) (cadr ur))
         lr (list (car ur) (cadr ll))
   )
   (command "pline" ll lr ur ul "Close")
   (command "ucs" "p")
   (princ)
)
```

This routine accomplishes the same task as the previous routine by performing the geometric calculations with the **sin** and **cos** AutoLISP functions. The result is correct only if the current UCS is parallel to the plane of the Text entity.

```
(defun C:TBOX2 ( / textent ang sinrot cosrot t1 t2 p0 p1 p2 p3 p4)
   (setq textent (entget (car (entsel "\nSelect text: "))))
   (setq p0 (cdr (assoc 10 textent))
         ang (cdr (assoc 50 textent))
         sinrot (sin ang)
         cosrot (cos ang)
         t1 (car (textbox textent))
         t2 (cadr (textbox textent))
         p1 (list (+ (car p0)
                     (- (* (car t1) cosrot) (* (cadr t1) sinrot)))
                  (+ (cadr p0)
                     (+ (* (car t1) sinrot) (* (cadr t1) cosrot))))
         p2 (list (+ (car p0)
                     (- (* (car t2) cosrot) (* (cadr t1) sinrot)))
                  (+ (cadr p0)
                     (+ (* (car t2) sinrot) (* (cadr t1) cosrot))))
         p3 (list (+ (car p0)
                     (- (* (car t2) cosrot) (* (cadr t2) sinrot)))
                  (+ (cadr p0)
                     (+ (* (car t2) sinrot) (* (cadr t2) cosrot))))
         p4 (list (+ (car p0)
                     (- (* (car t1) cosrot) (* (cadr t2) sinrot)))
                  (+ (cadr p0)
                     (+ (* (car t1) sinrot) (* (cadr t2) cosrot))))
   )
   (command "pline" p1 p2 p3 p4 "c")
   (princ)
)
```

Getting User Input

Several functions enable an AutoLISP application to interactively request data from the AutoCAD user.

The User-input (get*xxx*) Functions

Each user-input **get*xxx*** function pauses for the user to enter data of the indicated type and returns the value entered. The application can specify an optional prompt to display before the function pauses.

Table 2–1. Allowable input to the getxxx user-input functions

Function name	Type of user input
getint	An integer value from the prompt line
getreal	A real or integer value from the prompt line
getstring	A string from the prompt line
getpoint	A point value from the prompt line or selected from the screen
getcorner	A point value (the opposite corner of a box) from the prompt line or selected from the screen
getdist	A real or integer value (a distance) from the prompt line or determined by selecting points on the screen
getangle	An angle value (in the current angle format) from the prompt line or determined by selecting points on the screen
getorient	An angle value (in the current angle format) from the prompt line or determined by selecting points on the screen
getkword	A pre-defined keyword or its abbreviation from the prompt line

Note: The **getvar** and **getenv** functions are not user-input **get*xxx*** functions; they do not pause for user input.

The functions **getint**, **getreal**, and **getstring** all pause for user input, of the appropriate type, from the AutoCAD prompt line. Their built-in error checking mechanisms only allow them to return a value of the same type as that requested.

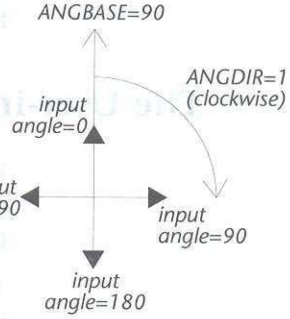
The **getpoint**, **getcorner**, and **getdist** functions pause for user input from the prompt line or from points selected on the graphics screen. These functions also filter out inappropriate responses. The **getpoint** and **getcorner** functions return 3D point values and **getdist** returns a real value.

Both **getangle** and **getorient** pause for input of an angle value from the prompt line or as defined by points selected on the graphics screen. For the **getorient** function, the zero angle is always to the right: "east" or "3 o'clock." For **getangle**, the zero angle is the value of ANGBASE, which can be set to any angle. Both **getangle** and **getorient** return an angle value (a real) in radians measured counterclockwise from a base (zero angle), for **getangle** equal to ANGBASE, and for **getorient** to the right. In response to these functions, the user can either enter a response at the prompt line or select points on screen.

Suppose ANGBASE is set to 90 degrees (north) and ANGDIR is set to 1 (clockwise direction for increasing angles). The following table shows what `getangle` and `getorient` return (in radians) for representative input values (in degrees).

Table 2–2. Possible return values from getangle and getorient

Input (degrees)	getangle	getorient
0	0.0	1.5708
–90	1.5708	3.14159
180	3.14159	4.71239
90	4.71239	0.0



The `getangle` function honours the settings of ANGDIR and ANGBASE. Thus you might use `getangle` to obtain a rotation amount for a Block insertion, since input of zero degrees always returns zero radians. The `getorient` function honours only ANGDIR. Thus you should use `getorient` to obtain such angles as the baseline angle for a Text entity. For example, given the above settings of ANGBASE and ANGDIR, for a line of text created at an angle of zero, `getorient` returns an angle value of 90.

The user-input functions take advantage of the error-checking capability of AutoCAD. Trivial errors (such as entering only a single number in response to `getpoint`) are trapped by AutoCAD and aren't returned by the user-input function. A prior call to `initget`, discussed later, provides additional filtering capabilities, lessening the need for extensive error checking.

The `getkword` function pauses for input of a keyword or its abbreviation. Keywords must be defined with the `initget` function before the call to `getkword`. All user-input functions (except `getstring`, for the obvious reason) can accept keyword values in addition to the values they normally return, provided `initget` has been called to define the keywords.

All user-input functions allow for an optional *`prompt`* argument. We recommend using this argument, rather than a prior call to the `prompt` or `princ` functions, to provide an application's prompt for user input. If a *`prompt`* argument is supplied with the call to the user-input function, that prompt is reissued in the case of invalid user input. If no *`prompt`* argument is supplied and the user enters incorrect information, the following message appears at the AutoCAD prompt line:

```
Try again:
```

This can be confusing since the original prompt might have scrolled out of the command prompt area.

Important: The AutoCAD user *cannot* typically respond to a user-input function by entering an AutoLISP expression. If your AutoLISP routine makes use of the `initget` function, arbitrary keyboard input is permitted to certain functions that can allow an AutoLISP statement as response to an AutoLISP-implemented command.

Control of User-input Function Conditions

The **initget** function establishes various options for use by the next **entsel**, **nentsel**, **nentselp**, or **getxxx** function (except **getstring**, **getvar**, and **getenv**). This function accepts two arguments, *bits* and *string*, both of which are optional. The *bits* argument specifies one or more control bits that enable or disable certain input values to the following user-input function call. The *string* argument can specify keywords that the following user-input function call will recognize.

Note: The control bits and keywords established by **initget** apply *only* to the next user-input function call; they are automatically discarded immediately afterward. The application doesn't have to call **initget** a second time to clear any special conditions.

Input Options for User-input Functions

The value of the *bits* argument restricts the types of user input to the following user-input function call. This reduces error checking by forcing the user to enter the desired type of information. These are some of the available bit settings: 1, disallows null input; 2, disallows input of 0 (zero); 4, disallows negative input. If these bit values are used with a following call to the **getint** function, the user is forced to enter an integer value greater than zero.

To set more than one condition at a time, simply add the values together (in any combination) to create a *bits* value between 0 and 255. If *bits* is not included or is set to zero, none of the control conditions apply to the next user-input function call. For a complete listing of available bit settings, see page 128.

Example

```
(initget (+ 1 2 4))
(getint "\nHow old are you? ")
```

This sequence requests AutoCAD to obtain the user's age. AutoCAD automatically displays an error message and repeats the prompt if the user attempts to enter a negative or zero value, type ↵ only, or enter a string (the **getint** function itself rejects any attempt to enter a value that is not an integer).

Keyword Options

The optional *string* argument specifies a list of keywords that will be recognized by the next user-input function call. The meaning of the keywords and the action to perform for each is the responsibility of the AutoLISP application.

The **initget** function allows keyword abbreviations to be recognized in addition to the full keywords. There are two methods for abbreviating keywords; both are discussed in the section "Keyword Specifications," on page 130. The user-input function returns a predefined keyword if the input from the user matches the spelling of a keyword (not case-sensitive), or if the user enters the abbreviation of a keyword.

Example

The following user-defined function shows a call to **getreal** preceded by a call to **initget** that specifies two keywords. The application checks for these keywords and sets the input value accordingly:

```
(defun C:GETNUM (/ num)
  (initget 1 "Pi Two-pi")
  (setq num (getreal "Pi/Two-pi/number: "))
  (cond ((eq num "Pi") pi)
        ((eq num "Two-pi") (* 2.0 pi))
        (T num)
  )
)
```

This **initget** call inhibits null input (*bits* = 1) and establishes a list of two keywords, "Pi" and "Two-pi". The **getreal** function is then used to obtain a real number, issuing the prompt:

Pi/Two-pi/number:

The result is placed in local symbol num. If the user enters a number, that number is returned by **C:GETNUM**. However, if the user enters the keyword **Pi** (or simply **P**), **getreal** returns the keyword Pi. The **cond** function detects this and returns the value of π in this case. The Two-pi keyword is handled similarly.

Note: You can also use **initget** to enable **entsel**, **nentsel**, and **nentselp** to accept keyword input (normally these functions expect the user to select an entity by picking a single point). For more information on these functions, see "Entity Name and Data Functions" on page 52, and the function descriptions on page 110 and page 138.

Arbitrary Keyboard Input

The **initget** function also allows arbitrary keyboard input to most of the ***getxxx*** functions. This input is passed back to the application as a string. An application using this facility can be written to permit the user to call an AutoLISP function at a ***getxxx*** function prompt.

Examples

These functions show a method for allowing AutoLISP response to a ***getxxx*** function call:

```
(defun C:ARBENTRY ( / pt1)                  Defines the function
  (initget 128)                              Sets arbitrary entry bit
  (setq pt1 (getpoint "\nPoint: "))          Gets value from user
  (if (= 'STR (type pt1))                    If it's a string value
    (setq pt1 (eval (read pt1)))             convert it to a symbol and try
                                             evaluating it as a function
    pt1                                      else, just return the value
  )
)
```

```
(defun ref ()
  (setvar "LASTPOINT" (getpoint "\nReference point: "))
  (getpoint "\nNext point: " (getvar "LASTPOINT"))
)
```

If the **C:ARBENTRY** and **REF** functions are both loaded into the drawing, then the following command sequence is perfectly acceptable.

```
Command: arbentry
Point: (ref)
Reference point: Select a point
Next point: @1,1,0
```

Conversions

The functions described in this section are utilities for converting data types and units.

For a sample function that converts from degrees to radians see the user defined **dtr** function on page 222.

String Conversions

The functions **rtos** (real to string) and **angtos** (angle to string) convert numeric values used in AutoCAD to string values that can be used in output or as textual data. The **rtos** function converts a real value and **angtos** converts an angle. The format of the result string is controlled by the value of AutoCAD system variables: the units and precision are specified by LUNITS and LUPREC for real (linear) values, and by AUNITS and AUPREC for angular values. For both functions, the dimensioning variable DIMZIN controls how leading and trailing zeros are written to the result string.

The following code fragments show some calls to **rtos** and the values returned (assuming the DIMZIN variable equals zero). Precision (the third argument to **rtos**) is set to 4 places in the first call and 2 places in the others.

```
(setq x 17.5)
(setq str "\nValue formatted as ")

(setq fmtval (rtos x 1 4))          Mode 1 = scientific
(princ (strcat str fmtval))         returns    Value formatted as 1.7500E+01

(setq fmtval (rtos x 2 2))          Mode 2 = decimal
(princ (strcat str fmtval))         returns    Value formatted as 17.50

(setq fmtval (rtos x 3 2))          Mode 3 = engineering
(princ (strcat str fmtval))         returns    Value formatted as 1'-5.50"

(setq fmtval (rtos x 4 2))          Mode 4 = architectural
(princ (strcat str fmtval)          returns    Value formatted as 1'-5 1/2"

(setq fmtval (rtos x 5 2))          Mode 5 = fractional
(princ (strcat str fmtval))         returns    Value formatted as 17 1/2
```

Note: When the UNITMODE system variable is set to one, specifying that units are displayed as they would be entered, the string returned by **rtos** differs for engineering (*mode* equals 3), architectural (*mode* equals 4), and fractional (*mode* equals 5) units. For example, the first two lines of the previous sample output would be the same, but the last three lines would appear as follows:

```
Value formatted as 1’5.50"
Value formatted as 1’5-1/2"
Value formatted as 17-1/2
```

You should note that since the **angtos** function takes the ANGBASE system variable into account, the code

```
(angtos (getvar "angbase"))
```

always returns "0".

Thus there is no AutoLISP function that returns a string version (in the current mode/precision) of either the amount of rotation of ANGBASE from true zero (East), or an arbitrary angle in radians.

To find the amount of rotation of ANGBASE from AutoCAD zero (East) or the size of an arbitrary angle, you can do one of the following:

1. Add the desired angle to the current ANGBASE, check to see if the absolute value of the result is greater than 2π, subtract 2π if it is (or add 2π if the result is negative), and then use the **angtos** function on the result.
2. Store the value of ANGBASE in a temporary variable, set ANGBASE to 0, evaluate the **angtos** function, and then set ANGBASE back to its original value.

Subtracting the result of (angtos 0) from 360 degrees (2π radians or 400grads) also yields the rotation of ANGBASE from zero.

The **distof** function is the complement of **rtos**, so the following calls, which use the strings generated in the previous examples, all return the same value, 17.5 (note the use of the backslash with modes 3 and 4).

```
(distof "1.7500E+01" 1)         Mode 1 = scientific

(distof "17.50" 2)              Mode 2 = decimal

(distof "1’-5.50\"" 3)          Mode 3 = engineering

(distof "1’-5 1/2\"" 4)         Mode 4 = architectural

(distof "17 1/2" 5)             Mode 5 = fractional
```

The following code fragments show similar calls to **angtos** and the values returned (still assuming that DIMZIN equals zero). Precision (the third argument to **angtos**) is set to 0 places in the first call, 4 places in the next three calls, and 2 places in the last.

```
(setq ang 3.14159) str2 "\nAngle formatted as ")

(setq fmtval (angtos ang 0 0))  Mode 0 = degrees
(princ (strcat str2 fmtval))    returns    Angle formatted as 180

(setq fmtval (angtos ang 1 4))  Mode 1 = deg/min/sec
(princ (strcat str2 fmtval))    returns    Angle formatted as 180d0'0"
```

`(setq fmtval (angtos ang 2 4)`	*Mode 2 = grads*	
`(princ (strcat str2 fmtval))`	*returns*	Angle formatted as 200.0000g
`(setq fmtval (angtos ang 3 4)`	*Mode 3 = radians*	
`(princ (strcat str2 fmtval))`	*returns*	Angle formatted as 3.1416r
`(setq fmtval (angtos ang 4 2)`	*Mode 4 = surveyor's*	
`(princ (strcat str2 fmtval))`	*returns*	Angle formatted as W

Note: The UNITMODE system variable also affects strings returned by **`angtos`** when it returns a string in surveyor's units (*mode* equals 4). If UNITMODE equals zero, the string returned can include spaces (for example, `"N 45d E"`); if UNITMODE equals one, the string contains no spaces (for example, `"N45dE"`).

The **`angtof`** function complements **`angtos`**, so the following calls all set the result argument to the same value, 3.14159.

`(angtof "180" 0)`	*Mode 0 = degrees*
`(angtof "180d0'0\"" 1)`	*Mode 1 = deg/min/sec*
`(angtof "200.0000g" 2)`	*Mode 2 = grads*
`(angtof "3.14159r" 3)`	*Mode 3 = radians*
`(angtof "W" 4)`	*Mode 4 = surveyor's*

Reminder: When you have a string specifying a distance in feet and inches or an angle in degrees, minutes, and seconds, you must use a backslash (\) to escape the double quote symbol (") so that it doesn't look like the end of the string. The previous examples of **`angtof`** and **`distof`** demonstrate this.

Real-world Units

The file *acad.unt* defines various conversions between real-world units such as miles/kilometres, Fahrenheit/Celsius, and so on. The function **`cvunit`** takes a value expressed in one system of units and returns the equivalent value in another system. The two systems of units are specified by strings containing expressions of units defined in *acad.unt* (see chapter 2 of the *AutoCAD Customization Manual* for more information about real-world units).

Example

If the current drawing units are engineering or architectural (feet and inches), the following routine converts a user-specified distance of inches into metres:

```
(defun C:I2M (/ eng_len metric_len eng metric)
  (princ "\nConverting inches to metres. ")
  (setq eng_len (getdist "\nEnter a distance in inches: "))
  (setq metric_len (cvunit eng_len "inches" "meters"))
  (setq eng (rtos eng_len 2 4) metric (rtos metric_len 2 4))
  (princ (strcat "\n\t" eng " inches = " metric " metres."))
  (princ)
```

The **cvunit** function will not convert between units whose dimensions are incompatible, such as an attempt to convert inches into grams.

Coordinate System Transformations

The **trans** function translates a point or a displacement from one coordinate system into another. It takes a point argument, *pt*, that can be interpreted as either a three-dimensional (3D) point or a 3D displacement vector, distinguished by a displacement argument called *disp*. The *disp* argument must be nonzero if *pt* is to be treated as a displacement vector; otherwise, *pt* is treated as a point.

A *from* argument specifies the coordinate system in which *pt* is expressed, and a *to* argument specifies the desired coordinate system. The following list describes the AutoCAD coordinate systems that can be specified by the *from* and *to* arguments:

WCS World Coordinate System: the "reference" coordinate system. All other coordinate systems are defined relative to the WCS, which never changes. Values measured relative to the WCS are stable across changes to other coordinate systems.

UCS User Coordinate System: the "working" coordinate system. The user specifies a UCS to make certain drawing tasks easier (or, in some cases, possible at all). All points passed to AutoCAD commands, including those returned from AutoLISP routines and external functions, are points in the current UCS (unless the user precedes them with a * at the Command: prompt). If you want your application to send coordinates in the WCS, ECS, or DCS to AutoCAD commands, you must first convert them to the UCS by calling the **trans** function.

ECS Entity Coordinate System. Point values returned by **entget** are expressed in this coordinate system, relative to the entity itself; such points are usually converted into the WCS, current UCS, or current DCS, according to the intended use of the entity. Conversely, points must be translated into an ECS before they are written to the database via **entmod** or **entmake**.

DCS Display Coordinate System. The coordinate system into which objects are transformed before they're displayed. The origin of the DCS is the point stored in the AutoCAD system variable TARGET and its *Z* axis is the viewing direction. In other words, a viewport is always a plan view of its DCS. These coordinates can be used to determine where something appears to the AutoCAD user.

When the *from* and *to* integer codes are 2 and 3, in either order, then 2 indicates the DCS for the current model space viewport, and 3 indicates the DCS for paper space (PSDCS). When the 2 code is used with an integer code other than 3 (or another means of specifying the coordinate system), then it is assumed to indicate the DCS of the current space, whether that is paper space or model space, and the other argument is also assumed to indicate a coordinate system in the current space.

PSDCS Paper Space DCS. This coordinate system can be transformed *only* to or from the DCS of the currently active model space viewport. This is essentially a 2D transformation, where the *X* and *Y* coordinates are always scaled and are offset if the `disp` argument is zero. The *Z* coordinate *is* scaled, but is never translated: it can thus be used to find the scale factor between the two coordinate systems. The PSDCS (integer code 2) can only be transformed into the current model space viewport: if the `from` argument equals 3, the `to` argument must equal 2, and vice versa.

Both the `from` and `to` arguments can specify a coordinate system in any of the following ways:

- An integer code that specifies the WCS, current UCS, or current DCS (of either the current viewport or paper space).
- An entity name, as returned by one of the entity name or selection set functions described in chapter 3. This specifies the ECS of the named entity.

  For planar entities, the ECS can differ from the WCS, as described in chapter 11 of the *AutoCAD Customization Manual.* If the ECS does not differ, conversion between ECS and WCS is an identity operation.
- A 3D extrusion vector is another method of specifying an entity's ECS. Extrusion vectors are always represented in World coordinates; an extrusion vector of (0,0,1) specifies the WCS itself.

Table 2–3. Coordinate system codes

Code	Coordinate system
0	World (WCS)
1	User (current UCS)
2	Display: DCS of current viewport when used with code 0 or 1 DCS of current model space viewport when used with code 3
3	Paper space DCS, PSDCS (used only with code 2)

Example

The following example translates a point from the WCS into the current UCS:

```
(setq pt ’(1.0 2.0 3.0))
(setq cs_from 0)                WCS
(setq cs_to 1)                  UCS
                                disp = 0 indicates that pt is a point:
(trans pt cs_from cs_to 0)
```

If the current UCS is rotated 90 degrees counterclockwise around the World *Z* axis, the call to **`trans`** returns a point (2.0,–1.0,3.0). On the other hand, if **`trans`** is called as follows:

```
(trans pt cs_to cs_from 0)      the result is  (-2.0,1.0,3.0)
```

Display Control

AutoLISP includes several functions for controlling the AutoCAD display, including both text and graphics screens. Some of these functions also prompt for, or depend on, input from the AutoCAD user.

Interactive Output

The basic output functions are **`prompt`**, **`princ`**, **`prin1`**, and **`print`**. The **`prompt`** function displays a message at the AutoCAD prompt line and returns `nil`; this should only be used when displaying a message to the screen. The **`princ`**, **`prin1`**, and **`print`** functions all display an expression (not necessarily a string) at the prompt line and return the expression, and optionally can send output to a file. The differences are as follows: **`princ`** prints control characters without expansion, while **`prin1`** and **`print`** expand characters with a leading backslash (\); **`print`** places a newline character before the expression and a space afterward.

The size of a string displayed by **`prompt`** should not exceed the length of the graphics screen's prompt line, which is typically no more than 80 characters.

Example

The following example demonstrates the differences between the four basic output functions and how they handle the same string of text (the two octal codes are DOS specific; see your operating system manual for octal codes that apply to your system, however the basic concept is the same regardless of the operating system):

```
(setq str "The \"allowable\" tolerance is \361 \254\"")

(prompt str)  prints       The "allowable" tolerance is ± 1/4"
              and returns  nil

(princ str)   prints       The "allowable" tolerance is ± 1/4"
              and returns  "The "allowable" tolerance is \361 \254""

(prin1 str)   prints       "The "allowable" tolerance is \361 \254""
              and returns  "The "allowable" tolerance is \361 \254""

(print str)   prints       (newline)
                           "The "allowable" tolerance is \361 \254"" (space)
              and returns  "The "allowable" tolerance is \361 \254""
```

The **`menucmd`** function provides control of the display of the graphics screen menus. This function activates one of the submenus of the current menu. It takes a string argument that consists of two parts, separated by an equals sign, in the following form:

```
"section=submenu"
```

where *`section`* specifies the menu section and *`submenu`* specifies which submenu to activate within that section. The allowed values of *`section`* are the same as in menu file submenu references.

The *submenu* argument can be a submenu label as defined in the current menu file, or a main section label (such as SCREEN). If *section* specifies a pull-down menu or the icon menu, *submenu* can be an asterisk (*): this causes the menu to be displayed (pull-down menus and the icon menu are not automatically displayed when they are activated). See chapter 6 of the *AutoCAD Customization Manual* for further information about menu files.

Note: The *string* argument should *not* include the dollar sign that introduces the comparable instructions in a menu file, and the *submenu* portion of *string* doesn't include the asterisks that precede submenu labels in the menu file.

Examples

The following **menucmd** function call causes the **OSNAPB submenu defined in the current menu file to appear on screen:

```
(menucmd "S=OSNAPB")
```

In a similar way the following call assigns the submenu **MY-BUTTONS to the ***BUTTONS1 menu and activates it.

```
(menucmd  "B1=MY-BUTTONS")
```

The following code forces the Cursor menu (defined as ***POP0) to appear on screen:

```
(menucmd "P0=POP0")    Initializes the ***POP0 menu
(menucmd "P0=*")       Displays it
```

The following call displays the pull-down menu currently initialized in the P1 (first pull-down menu) location:

```
(menucmd "P1=*")
```

Using "P1=*" without previously initializing the desired menu usually works fine but can result in unexpected behaviour. Although you can initialize virtually any menu at a pull-down or cursor menu location, it is best to use only menus specifically designed for that menu area. If you have a submenu called **MORESTUFF, you can display it at the P1 location by using the following code:

```
(menucmd "P1=MORESTUFF")    Initializes the **MORESTUFF menu
(menucmd "P1=*")            Displays it in the P1 location
```

This menu remains in this location until you replace it by initializing another menu, as in the following:

```
(menucmd "P1=POP1")
```

If your menu makes use of the disabling (graying-out) and marking features you can retrieve and change the state of a menu label with the **menucmd** function. The following call retrieves the current state of the fourth label in the pull-down menu P2:

```
(menucmd "P2.4=#?")   if disabled returns        "P2.4=~"
```

These function calls enable and disable that same label:

```
(menucmd "P2.4=")     Enables the label
(menucmd "P2.4=~")    Disables the label
```

You can also place and remove marks to the left of menu labels in a similar manner. See chapter 6 in the *AutoCAD Customization Manual* for more information on menus.

Not only can an AutoLISP function enable and disable menu labels, it can also modify the text displayed in the label by placing a DIESEL string expression in the label itself. Since DIESEL only accepts strings as input you can pass information to the DIESEL expression through a USERS1–5 system variable that has been set to a value returned by your function. See chapter 8 of the *AutoCAD Customization Manual* for more information on the use of DIESEL expressions in menus.

The **menucmd** function also lets you evaluate DIESEL string expressions within an AutoLISP function. The following routine returns the current time:

```
(defun C:CTIME ( / ctim)
   (setq ctim (menucmd "M=$(edtime,$(getvar,date),H:MMam/pm)"))
   (princ (strcat "\nThe current time is " ctim ))
   (princ)
)
```

See chapter 8 of the *AutoCAD Customization Manual* for more information on the use of DIESEL expressions with AutoLISP and a catalog of DIESEL functions.

Control of Graphics and Text Screens

To switch between the text and graphics screens on single-screen AutoCAD installations, an AutoLISP application can call **graphscr** to display the graphics screen, or **textscr** to display the text screen. These functions are equivalent to the AutoCAD commands GRAPHSCR and TEXTSCR, or to toggling the Flip Screen function key. The function **textpage** is like **textscr**, but clears the text screen before displaying it (as the AutoCAD STATUS and HELP commands do).

The **redraw** function is similar to the AutoCAD REDRAW command, but provides more control over what is displayed: it can not only redraw the entire graphics screen, but also can specify a single entity to be either redrawn or undrawn (i.e., blanked out). If the entity is a complex entity such as a Polyline or Block, **redraw** can draw (or undraw) either the entire entity, or only its header. The **redraw** function can also highlight and unhighlight specified entities.

Control of Low-level Graphics and User Input

These functions provide direct access to the AutoCAD graphics screen and input devices. They enable AutoLISP applications to use some of the display and user interaction facilities built in to AutoCAD.

The function **grclear** clears the current viewport without affecting the prompt, status, or menu areas of the graphics screen. The **grtext** function displays text directly in the status or menu areas, with or without highlighting. The **grdraw** function draws a vector in the current viewport, with control over colour and highlighting. The **grvecs** function draws multiple vectors. Finally, **grread** returns "raw" user input, whether from the keyboard or the pointing device (mouse or digitizer); if the call to **grread** enables tracking, the function returns a digitized coordinate that can be used for such things as dragging.

Caution: Because these functions depend on code in AutoCAD, their operation can be expected to change from release to release: there is no guarantee that applications calling these functions will be upward compatible. Also, they depend on the current hardware configuration: in particular, applications that call **grtext** and **grread** are not likely to work the same on all configurations unless the developer is very careful to use them as described (see chapter 4, page 124 through page 126) and avoid hardware-specific features. Finally, because they are low-level functions, they do almost no error reporting and can mess up the graphics screen display (see the example for a way to fix this).

Example

The following sequence reverses any damage to the graphics screen display caused by incorrect calls to **grclear**, **grtext**, **grdraw**, or **grvecs**:

```
(grtext)                        Restores standard text
(redraw)
```

Tablet Calibration

AutoCAD users with a digitizing tablet can calibrate the tablet by using the TABLET command, as described in chapter 4 of the *AutoCAD Reference Manual*. The **tablet** function lets applications manage calibration by setting them directly and by saving calibration settings for future use.

The first argument to the **tablet** function is an integer `code`. If `code` is equal to 0, the function returns the current calibration. If `code` is equal to 1, the calibration is set according to the remaining arguments. Calibrations are expressed as four three-dimensional points (in addition to the `code`). The first three of these points—`row1`, `row2`, and `row3`—are the three rows of the tablet's transformation matrix. The fourth point, `direction`, is a vector that is normal to the plane the tablet's surface is assumed to lie in (expressed in WCS, the World Coordinate System). When calibrating is done with the TABLET command, the tablet's surface is assumed to lie in the *XY* plane of the current UCS.

Note: The system variable TABMODE controls whether Tablet mode is On (1) or Off (0). You can control it by using the **setvar** function.

Examples

The following sample routine retrieves the current tablet calibration and stores it in the symbol `tcal`.

```
(defun C:TABGET ( )
  (setq tcal (tablet 0))
  (if (not tcal)
    (princ "\nCalibration not obtainable ")
    (princ "\nConfiguration saved, use TABSET to retrieve calibration.")
  )
  (princ)
)
```

If the above routine was successful, the symbol `tcal` now contains the list returned by the tablet function. This list might appear as follows:

```
(1 (0.00561717 -0.000978942 -7.5171)
   (0.000978942 0.00561717 -9.17308)
   (0.0 0.0 1.0)
   (0.0 0.0 1.0)
)
```

To reset the calibration to the values retrieved by the previous routine, you can use the **C:TABSET** routine:

```
(defun C:TABSET ( )
  (if (not (apply 'tablet tcal))
    (princ "\nUnable to reset calibration. ")
    (progn
      (princ "\nTablet calibration reset. ")
      (setvar "tabmode" 1)                      Sets tablet mode On
      (if (= (getvar "tabmode") 0)
        (princ "\nUnable to turn on tablet mode ")
      )
    )
  )
  (princ)
)
```

The transformation matrix passed as *row1*, *row2*, and *row3* is a 3×3 transformation matrix meant to transform a two-dimensional point. The 2D point is expressed as a column vector in *homogeneous coordinates* (by appending 1.0 as the third element), so the transformation looks like this:

$$\begin{bmatrix}X'\\Y'\\D'\end{bmatrix} = \begin{bmatrix}M_{00} & M_{01} & M_{02}\\M_{10} & M_{11} & M_{12}\\M_{20} & M_{21} & 1.0\end{bmatrix}\begin{bmatrix}X\\Y\\1.0\end{bmatrix}$$

The calculation of a point is similar to the 3D case. AutoCAD transforms the point by using the following formulas:

$$X' = M_{00}X + M_{01}Y + M_{02}$$

$$Y' = M_{10}X + M_{11}Y + M_{12}$$

$$D' = M_{20}X + M_{21}Y + 1.0$$

To turn the resulting vector back into a 2D point, the first two components are divided by the third, the scale factor D', yielding the point $(X'/D',Y'/D')$.

For projective transformations, the most general case, `tablet` actually does the full calculation. But for affine and orthogonal transformations, M_{20} and M_{21} are both zero, so D' would be 1.0. The calculation of D' and the division are omitted: the resulting 2D point is simply (X',Y').

As the previous paragraph implies, an affine transformation is a special, uniform case of a projective transformation. An orthogonal transformation is a special case of an affine transformation: not only are M_{20} and M_{21} zero, but $M_{00} = M_{11}$ and $M_{10} = -M_{01}$.

Note: When you set a calibration, the list returned *won't* equal the list provided if the *direction* wasn't normalized: AutoCAD normalizes the direction vector before it returns it. Also, it ensures that the third element in the third column (*row3*[Z]) is equal to 1. This situation shouldn't arise if you set the calibration using values retrieved from AutoCAD via **tablet**. However, it can happen if your program calculates the transformation itself.

Wild Card Matching

The **wcmatch** function enables applications to compare a string to a wild card pattern. This facility can be used when building a selection set (in conjunction with **ssget** as described in the section "Handling Selection Sets" on page 45),and when retrieving extended entity data by application name (in conjunction with **entget** as described in "Retrieving Extended Entity Data" on page 66).

The **wcmatch** function compares a single string to a pattern; it returns T if the string matches the pattern, and nil if it does not. The wild card patterns are similar to the *regular expressions* used by many system and application programs. In the pattern, alphabetic characters and numerals are treated literally; brackets can be used to specify optional characters or a range of letters or digits; a question mark (?) matches a single character and an asterisk (*) matches a sequence of characters; and certain other special characters have special meanings within the pattern. See the description of **wcmatch** in chapter 4, page 166, for more details.

Examples

These examples assume that a string variable called matchme has been declared and initialized as follows:

```
(setq matchme "this is a string - test1 test2 test3 the end")
```

The following call checks whether matchme *begins* with the four characters "this":

```
(wcmatch matchme "this*")              returns      T
```

The following call illustrates the use of brackets in the pattern. In this case, **wcmatch** returns T if matchme contains "test4", "test5", "test6", or "test9" (note the use of the "*" character at the beginning and end of the search pattern, allowing the desired portion to be located anywhere in the string):

```
(wcmatch matchme "*test[4-69]*")       returns      nil
```

however

```
(wcmatch matchme "*test[4-61]*")       returns      T
```

since the string contains "test1".

The pattern string can specify multiple patterns, separated by commas. The following call returns T if matchme equals "ABC", if it *begins* with "XYZ", or if it *ends* with "end".

```
(wcmatch matchme "ABC,XYZ*,*end")      returns      T
```

Chapter 3

Selection Set, Entity, and Symbol Table Functions

Most AutoLISP functions that handle selection sets and entities identify a set or entity by its *name,* an alphanumeric code assigned and maintained by AutoCAD. Before it can manipulate a selection set or entity, an AutoLISP application must first obtain the current name of the set or entity by calling one of the functions that return a selection set or entity name. Examples of these functions appear in the following sections.

Important: Selection-set and entity names are *volatile;* they apply only to the current drawing session.

For selection sets, valid only in the current session, the volatility of names poses no problem, but it does for entities, since they are saved in the drawing database. An application that must refer to the same entities in the same drawing (or drawings) at different times can use entity *handles,* described in the section "Entity Handles and Their Use" on page 53.

Handling Selection Sets

The **ssget** function provides the most general means of creating a selection set. It can create a selection set in one of the following ways:

- Explicitly specifying the entities to select by using the Last, Previous, Window, Implied, WPolygon, Crossing, CPolygon, or Fence options (as in interactive AutoCAD use), or by specifying a single point. The entire database can also be selected.
- Prompting the user to select objects.

You can use *filtering* with either of the previous groups of options. This lets you specify a list of attributes and conditions that the selected entities must match.

The first argument to **ssget** is a string describing which selection option to use. The next two arguments, `pt1` and `pt2`, specify point values for the relevant options (they should be left out if they don't apply). A point list, `pt-list`, must be provided as an argument to the selection methods that allow selection by Polygons (i.e., Fence, Crossing Polygon, and Window Polygon). The last argument, `filter-list`, is optional. If `filter-list` is supplied, it specifies the list of entity field values that are used in filtering. Selection filters are described in more detail in "Selection Set Filter Lists" on page 47. The following table summarizes the available mode values and the arguments used with each (a *filter-list* can be used as an additional argument to all of the selection methods listed):

Table 3–1. Selection options for ssget

Mode	Selection method	Prototypes
none	User selection or single-point selection (if *pt1* is specified)	`(ssget)` or `(ssget` *pt1*`)`
"L"	Last created entity visible on screen	`(ssget "L")`
"P"	Previous selection set	`(ssget "P")`
"I"	Implied selection set (previous set created with PICKFIRST mode On)	`(ssget "I")`
"W"	Window selection	`(ssget "W"` *pt1 pt2*`)`
"C"	Crossing selection	`(ssget "C"` *pt1 pt2*`)`
"F"	Fence (open polygon) selection	`(ssget "F"` *pt-list*`)`
"CP"	Crossing Polygon selection	`(ssget "CP"` *pt-list*`)`
"WP"	Window Polygon selection	`(ssget "WP"` *pt-list*`)`
"X"	Selects *all* entities in drawing	`(ssget "X")`

Caution: If *mode* "X" is specified and a *filter-list* is not provided, **ssget** selects *all* entities in the database, including layers that are off, frozen, and out of the visible screen.

Examples

The following code shows some representative calls to **ssget**:

`(setq pt1 '(0.0 0.0 0.0)`
`      pt2 '(5.0 5.0 0.0)`
`      pt3 '(4.0 1.0 0.0)`
`      pt4'(2.0 6.0 0.0)`
`)`
	Sets pt1, pt2, pt3, *and* pt4 *to point values*

`(setq ss1 (ssget))`	*Asks the user for a general entity selection and places those items in a selection set*
`(setq ss1 (ssget "P"))`	*Creates a selection set of the most recently selected objects*
`(setq ss1 (ssget "L"))`	*Creates a selection set of the last entity added to the database that is visible on screen*
`(setq ss1 (ssget pt2))`	*Creates a selection set of an entity passing through point (5,5)*
`(setq ss1 (ssget "W" pt1 pt2))`	*Creates a selection set of the entities inside the window from (0,0) to (5,5)*
`(setq ss1 (ssget "F" (list pt2 pt3 pt4)))`	*Creates a selection set of the entities crossing the fence defined by the points (5,5), (4,1), and (2,6)*
`(setq ss1 (ssget "WP" (list pt1 pt2 pt3)))`	*Creates a selection set of the entities inside the polygon defined by the points (0,0),(5,5), and(4,1)*
`(setq ss1 (ssget "X"))`	*Creates a selection set of all entities in the database*

When an application has finished using a selection set, it is important to release it from memory. This can be done by setting it to `nil`.

```
(setq ss1 nil)
```

Important: An AutoLISP application cannot have more than 128 selection sets open at once. The limit is determined by many factors and might be slightly lower on your system. When the limit is reached, AutoCAD refuses to create more selection sets. Attempting to simultaneously manage a large number of selection sets is not recommended. Instead, you should keep only a reasonable minimum number of sets open at a time and set unneeded selection sets to `nil` as soon as possible. If the maximum number of selection sets is reached, you must call **`gc`** (see "Node Space" on page 177 for information on garbage collection) before another **`ssget`** will work.

Selection Set Filter Lists

An entity filter list is an association list that uses DXF group codes (see appendix B for a list of group codes) in the same format as a list returned by **`entget`** (described later in this chapter). The **`ssget`** function recognizes all group codes except entity names (group –1), handles (group 5), and extended entity data codes (≥1000). If an invalid group code is used in a *`filter-list`*, it is ignored by **`ssget`**. To search for entities with extended data, use the –3 code as described in "Filtering for Extended Entity Data" on page 49.

When a *`filter-list`* is provided as the last argument to **`ssget`**, the function scans the selected entities and creates a selection set that contains the names of all main entities matching the specified criteria. For example, using this mechanism, you can obtain a selection set that includes all entities of a given type, on a given layer, or of a given colour.

The *`filter-list`* specifies which property (or properties) of the entities are to be checked, and what values constitute a match.

Examples

The four examples below demonstrate methods of using a *`filter-list`* with different *`mode`* selection options.

```
(setq ss1 (ssget
   '((0 . "TEXT")))
```

Asks the user for general entity selection but adds Text entities only to the selection set

```
(setq ss1 (ssget "P"
   '((0 . "LINE")))
)
```

Creates a selection set of the most recently selected objects that are also Line entities

```
(setq ss1 (ssget "W" pt1 pt2
   '((8 . "FLOOR9")))
)
```

Creates a selection set of all entities inside the window that are also on layer FLOOR9

```
(setq ss1 (ssget "X"
   '((0 . "CIRCLE")))
)
```

Creates a selection set of all entities in the database that are Circle entities

If both the code and the desired value are known, the list may be quoted as shown previously. If either is specified by a variable, the list must be constructed (using the **list** and **cons** functions).

```
(setq lay_name "FLOOR3")
(setq ss1
   (ssget "X"                      Creates a selection set of all entities in the database
      (list (cons 8 lay_name))     that are on layer FLOOR3
   )
)
```

If the *filter-list* specifies more than one property, an entity is included in the selection set only if it matches *all* specified conditions. For instance (continuing the previous example):

```
(ssget "X" (list (cons 0 "CIRCLE")(cons 8 lay_name)(cons 62 1)))
```

This selects only Circle entities on layer FLOOR3 that are the colour red. This type of test performs a Boolean AND operation. Additional tests for entity properties are discussed in "Logical Grouping of Filter Tests" on page 49.

The **ssget** function filters a drawing by scanning the selected entities and comparing the fields of each main entity against the specified filtering list. If an entity's properties match *all* specified fields in the filtering list, it is included in the returned selection set. Otherwise, the entity is not included in the selection set. The **ssget** function returns nil if no entities from those selected match the specified filtering criteria.

Caution: The meaning of certain group codes can differ from entity to entity, and not all group codes are present in all entities. If a particular group code is specified in a filter, entities not containing that group code will be excluded from the selection set that **ssget** returns.

Note: When **ssget** filters a drawing, the selection set it retrieves might include entities from both paper space and model space. However, when the selection set is passed to an AutoCAD command, only entities from the space that's currently in effect are used (the space to which an entity belongs is specified by the value of its 67 group, as described in appendix B of this manual and chapter 11 of the *AutoCAD Customization Manual*).

Wild Card Patterns in Filter Lists

Symbol names specified in filtering lists—the entity type (0), Block name (2), DIMSTYLE name (3), Linetype (6), Text style (7) and Layer name (8)—can include wild card patterns. The wild card patterns recognized by **ssget** are the same as those recognized by the function **wcmatch**, and are described in the section "Wild Card Matching" on page 43, and in the description of **wcmatch** in chapter 4, page 166.

Caution: When filtering for anonymous Blocks, you must *escape* the * character with a reverse quote (`) because the * is read by **ssget** as a wild card character. For example, you can retrieve an anonymous Block named *U2, with:

```
(ssget "X" ’((2 . "`*U2")))
```

Filtering for Extended Entity Data

Using the **ssget** *filter-list*, you can select all entities containing extended entity data for a particular application (see "Notes on Extended Entity Data" on page 63). This is done using the –3 group code, as in:

```
(ssget "X" ’((0 . "CIRCLE") (-3 ("APPNAME"))))
```

This would select all circles that include extended data for the "APPNAME" application. If more than one application name is included in the –3 group's list, an AND operation is implied and the entity must contain extended data for all of the specified applications. Thus

```
(ssget "X" ’((0 . "CIRCLE") (-3 ("APP1")("APP2"))))
```

would select all circles with extended data for both the "APP1" and "APP2" applications. Wild card matching is permitted, so either:

```
(ssget "X" ’((0 . "CIRCLE") (-3 ("APP[12]"))))
```

or

```
(ssget "X"’((0 . "CIRCLE")(-3 ("APP1,APP2"))))
```

would select all circles with extended data for either or both of these applications.

Relational Tests

Unless otherwise specified, an "equals" test is implied for each item in the *filter-list*. For numeric groups (integers, reals, points, and vectors) you can specify other relations by including a special –4 group code that specifies a relational operator. The value of a –4 group is a string indicating the test operator to be applied to the next group in the *filter-list*. See "Relational Tests" on page 155 for further information.

Example

The following selects all circles with radius (group code 40) greater than or equal to 2.0:

```
(ssget "X" ’((0 . "CIRCLE") (-4 . ">=") (40 . 2.0)))
```

Logical Grouping of Filter Tests

In addition to the relational operators described previously you can also test groups by creating nested Boolean expressions that use logical grouping operators (shown on page 156).

The grouping operators are specified by –4 groups, like the relational operators. They are paired and must be balanced correctly in the filter list or the **ssget** call will fail.

An example of grouping operators in a filter list follows:

```
(ssget "X" '((-4 . "<OR")
              (-4 . "<AND")
                (0 . "CIRCLE")
                (40 . 1.0)
              (-4 . "AND>")
              (-4 . "<AND")
                (0 . "LINE")
                (8 . "ABC")
              (-4 . "AND>")
            (-4 . "OR>")
           )
)
```

This selects all circles with radius 1.0 plus all lines on layer "ABC".

Note: The grouping operators aren't case-sensitive; you can also use their lowercase equivalents.

Grouping operators are not allowed within the –3 group itself. As discussed previously in "Filtering for Extended Entity Data," multiple application names specified in a –3 group use an implied AND operator. If you want to test for extended entity data using other grouping operators, you can do so by specifying separate –3 groups and grouping them as desired. To select all circles having extended data for either application "APP1" or "APP2" but not both, you would use:

```
(ssget "X" '((0 . "CIRCLE")
            (-4 . "<XOR")
              (-3 ("APP1"))
              (-3 ("APP2"))
            (-4 . "XOR>")
          )
)
```

You can simplify the coding of frequently used grouping operators by setting them equal to a symbol. The previous example could be rewritten as follows (notice that in this example you must explicitly quote each list):

```
(setq <xor '(-4 . "<XOR")
     xor> '(-4 . "XOR>")
)
(ssget "X" (list '(0 . "CIRCLE")
                   <xor
                     '(-3 ("APP1"))
                     '(-3 ("APP2"))
                 xor>
          )
)
```

As you can see, this method might not be sensible for short pieces of code, but can be beneficial in larger applications.

Selection Set Manipulation

Once a selection set has been created, entities can be added to it or removed from it with the functions **ssadd** and **ssdel**, which are similar to the Add and Remove options when AutoCAD has interactively prompted the user to Select objects: or Remove objects:.

The **ssadd** function can also be used to create a new selection set, as shown in the following example.

Example

The following code fragment creates a selection set that includes the first and last entities in the current drawing (**entnext** and **entlast** are described later in this chapter).

```
(setq fname (entnext))                     Gets first entity in the drawing
(setq lname (entlast))                     Gets last entity in the drawing
(if (not fname)
  (princ "\nNo entities in drawing. ")
  (progn
    (setq ourset (ssadd fname))            Creates a sel. set of the first entity
    (ssadd lname ourset)                   Adds the last entity to the same sel. set
  )
)
```

The example will run correctly even if there is only one entity in the database (in which case, both **entnext** and **entlast** set their arguments to the same entity name). If **ssadd** is passed the name of an entity already in the selection set, it simply ignores the request and does *not* report an error.

The following function removes the first entity from the selection set created in the previous example:

```
(ssdel fname ourset)
```

If there is more than one entity in the drawing (that is, if `fname` and `lname` are not equal), the selection set `ourset` now contains only `lname`, the last entity in the drawing.

The function **sslength** returns the number of entities in a selection set, and **ssmemb** tests whether a particular entity is a member of a selection set. Finally, the function **ssname** returns the name of a particular entity in a selection set, using an index into the set (entities in a selection set are numbered from 0).

Example

The following code shows a few calls to **ssname**:

```
(setq sset (ssget))                        Creates the selection set (by prompting
                                           the user)
(setq ent1 (ssname sset 0))                Gets the name of first entity in sset
(setq ent4 (ssname sset 3))                Gets the name of the fourth entity in sset
(if (not ent4)
  (princ "\nNeed to select at least four entities. ")
)
(setq ilast (sslength sset))               Finds index of the last entity in sset
                                           Gets the name of the last entity in sset
(setq lastent (ssname sset (1- ilast)))
```

Note: Regardless of how entities have been added to a selection set, the set *never* contains duplicate entities. If the same entity is added more than once, the later additions are simply ignored. Because of this, **sslength** accurately returns the number of *distinct* entities in the specified selection set.

Entity Name and Data Functions

Entity-handling functions are organized into two categories: functions that retrieve the name of a particular entity, either by searching the database or prompting the AutoCAD user, and functions that retrieve or modify entity data.

Entity Name Functions

To operate on an entity, an AutoLISP application must obtain its name for use in subsequent calls to the entity data or selection set functions. Two of the functions described in this section, **entsel** and **nentsel**, return not only the entity's name, but also additional information for the application's use.

Both functions require the AutoCAD user to select an entity interactively by picking a point on the graphics screen: all of the other entity name functions can retrieve an entity even if it is not visible on screen or is on a frozen layer. The **entsel** function prompts the AutoCAD user to select an entity by picking a point on the graphics screen; **entsel** returns both the entity name and the value of the point that was selected. Some entity operations require knowledge of the point by which the entity was selected; examples from the set of existing AutoCAD commands include BREAK, TRIM, and EXTEND. The **nentsel** function is discussed in detail in the section, "Entity Context and Coordinate Transform Data" on page 54. These functions honour keywords if they are preceded by a call to **initget**; see page 128 for more information on **initget**.

The **entnext** function retrieves entity names sequentially. If **entnext** is called with no arguments, it returns the name of the first entity in the drawing database; if its argument is the name of an entity in the current drawing, it returns the name of the succeeding entity.

Example

This code fragment illustrates how **ssadd** can be used in conjunction with **entnext** to create selection sets and add members to an existing set.

```
(setq e1 (entnext))
(if (not e1)                            Sets e1 to name of first entity
  (princ "\nNo entities in drawing. ")
  (progn
    (setq ss (ssadd))                   Sets ss to a null selection set
    (ssadd e1 ss)                       Returns selection set ss with e1 added
    (setq e2 (entnext e1))              Gets entity following e1
    (ssadd e2 ss)                       Adds e2 to selection set ss
  )
)
```

The **entlast** function retrieves the name of the last entity in the database. The last entity is the most recently created main entity, so **entlast** can be called to obtain the name of an entity that has just been created via a call to **command**.

You can set the entity name returned by **entnext** to the same variable name passed to this function. This will essentially "bump" or "walk" a single entity name variable through the database. For example:

```
(setq one_ent (entnext))                       Gets name of first entity
(while one_ent
    .
    .                                          Processes new entity
    .
    (setq one_ent (entnext one_ent))
)                                              Value of one_ent is now nil
```

Entity Handles and Their Use

The **handent** function retrieves the name of an entity with a specific *handle*. Entity handles must be enabled in the current drawing: handles are controlled by the AutoCAD HANDLES command, and the system variable HANDLES, which equals 1 if they're enabled, 0 if disabled. *Like* entity names, handles are unique within a drawing. *Unlike* entity names, an entity's handle is constant throughout its life (provided the HANDLES command is not used to Destroy all handles in a database). AutoLISP applications that manipulate a specific database can use **handent** to obtain the current name of an entity they must use.

Example

The following code fragment uses **handent** to obtain an entity name and print it out:

```
(if (not (setq e1 (handent "5a2")))
    (princ "\nNo entity with that handle exists. ")
    (princ e1)
)
```

In one particular editing session, this code might print out:

```
<Entity name: 60004722>
```

In another editing session with the same drawing, the example might print an entirely different number. But in both cases, the code would be accessing the same entity.

The **handent** function has an additional use: entities that have been deleted from the database (via **entdel**, described in the following section) aren't purged until the current drawing ends. This means that **handent** can recover the names of deleted entities, which can then be restored to the drawing by a second call to **entdel**.

Note: When handles are On in a drawing, they are provided for Block definitions, including Block subentities, as well as for extended entity data. See the *AutoCAD Reference Manual* for more information.

Entities in drawings that are cross-referenced via XREF Attach are not actually part of the current drawing: their handles are unchanged, but cannot be accessed by **handent**. However, when drawings are combined via INSERT,

INSERT *, XREF Bind (XBIND), or partial DXFIN, the handles of entities in the incoming drawing are lost (provided they were present in the first place), and incoming entities are assigned new handle values to ensure that each handle in the current drawing remains unique.

Note: Extended entity data, described later in this chapter, can include entity handles for saving relational structures in a drawing. Sometimes, these handles also require translation or maintenance. See the section "Handles in Extended Entity Data" on page 68.

Entity Context and Coordinate Transform Data

The **nentsel** and **nentselp** functions are similar to **entsel**, except that they return two additional values which are meant to facilitate handling of entities that are *nested* within block references.

Important: Another difference between these functions is that when the user responds to a **nentsel** call by picking a complex entity or a complex entity is selected by **nentselp**, these functions return the entity name of the selected subentity and *not* the complex entity's header, as **entsel** does. For example, when the user picks a Polyline, **nentsel** returns a Vertex subentity instead of the Polyline header. To retrieve the Polyline header, the application must use **entnext** to walk forward to the Seqend subentity, then obtain the name of the header from the Seqend subentity's –2 group. The same applies when the user selects Attributes in a nested block reference. The **nentselp** function is generally preferred to **nentsel** since it returns a transformation matrix of the same format as that returned by **grvecs**.

The first of the additional elements returned by **nentsel** is 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 (this refers to Xrefs also) containing 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.

The second additional element is a list containing the entity name of the Block that contains the selected entity. If the selected entity is contained in a nested Block (a Block within a Block), the list additionally contains the entity names of all Blocks in which the selected entity is nested, starting with the innermost Block and continuing outward until the name of the outermost Block that was inserted in the drawing is reported.

```
(<Entity Name: ename1>              Name of entity
   (Px Py Pz)                       Pick point
   (   (X0 Y0 Z0)                   Model to World
       (X1 Y1 Z1)                   Transformation Matrix
       (X2 Y2 Z2)
       (X3 Y3 Z3)
   )
   (<Entity name: ename2>           Name of most deeply nested Block
             .                      containing selected entity
             .
             .
     <Entity name: enamen>)         Name of outermost Block
)                                   containing selected entity
```

For the following example, assume that the current coordinate system is the WCS. Create a Block named SQUARE consisting of four lines:

```
Command: line
From point: 1,1
to point: 3,1
to point: 3,3
to point: 1,3
to point: c
Command: block
Block name (or ?): square
Insertion base point: 2,2
Select objects: Select the four lines you just drew.
Select objects: ↵
```

Then insert the Block in a UCS rotated 45 degrees about the *Z* axis:

```
Command: ucs
Origin/ZAxis/3point/Entity/View/X/Y/Z/Prev/Restore/Save/Del/?/<World>: z
Rotation angle about Z axis <0>: 45
Command: insert
Block name (or ?): square
Insertion point: 7,0
X scale factor <1> / Corner / XYZ: ↵
Y scale factor (default=X): ↵
Rotation angle: ↵
```

Use **nentsel** to select the lower-left side of the square.

```
(setq ndata (nentsel))
```

This sets the symbol ndata equal to a list similar to:

```
(<Entity Name: 400000a0>                        Entity name
   (6.46616 -1.0606 0.0)                        Pick point
   (   (0.707107 0.707107 0.0)                  Model to World
       (-0.707107 0.707107 0.0)                 transformation matrix
       (0.0 -0.0 1.0)
       (4.94975 4.94975 0.0)
   )
   (<Entity name: 6000001c>)                    Name of Block
)                                               containing selected entity
```

Once you've obtained the entity name and the Model to World Transformation Matrix is obtained, you can transform the entity definition data points from the MCS to the WCS. Use **entget** and **assoc** on the entity name to obtain the desired definition points expressed in MCS coordinates. Then pass the points and the Model to World Transformation Matrix data (obtained in the first **nentsel** call) to the formulas below.

If the selected entity is not a nested entity, the transformation matrix is simply set to the identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

The following equations show how to transform a point or vector:

$$X' = XM_{00} + YM_{10} + ZM_{20} + M_{30}$$

$$Y' = XM_{01} + YM_{11} + ZM_{21} + M_{31}$$

$$Z' = XM_{02} + YM_{12} + ZM_{22} + M_{32}$$

The M_{ij}, where $0 \le i, j \le 2$, are the Model to World Transformation Matrix coordinates, X, Y, and Z is the entity definition data point expressed in MCS coordinates, and X', Y', and Z' is the resulting entity definition data point expressed in WCS coordinates.

Note: To transform a vector rather than a point, don't add in the translation vector $\begin{bmatrix} M_{30} & M_{31} & M_{32} \end{bmatrix}$ (from the fourth column of the transformation matrix).

This example illustrates how to obtain the MCS start point of a line (group code 10) contained in a Block definition. The statement

```
(setq edata (assoc 10 (entget (car ndata))))
```

stores the entity data (using the entity name obtained with `nentsel` earlier) in the symbol `edata` and returns:

```
(10 -1.0 1.0 0.0)
```

The statement

```
(setq matrix (caddr ndata))
```

stores the Model to World Transformation Matrix sublist in the symbol `matrix` and returns:

```
( (0.707107 0.707107 0.0)         X transformation
  (-0.707107 0.707107 0.0)        Y transformation
  (0.0 -0.0 1.0)                  Z transformation
  (4.94975 4.94975 0.0)           Displacement from WCS origin
)
```

Apply the transformation formula for X' to change the X coordinate of the start point of the line from an MCS coordinate to a WCS coordinate. Store the results in the symbol `answer`

```
(setq answer
    (+                                                     add:
        (* (car (nth 0 matrix)) (cadr edata))              M00 * X
        (* (car (nth 1 matrix)) (caddr edata))             M10 * Y
        (* (car (nth 2 matrix)) (cadddr edata))            M20 * Z
        (car (nth 3 matrix))                               M30
    )
)
```

which returns 3.53553, the WCS *X* coordinate of the start point of the selected line.

Entity Data Functions

The functions described in this section operate on entity data and can be used to modify the current drawing database.

The `entdel` function deletes a specified entity. The entity is not actually purged from the database until the end of the current drawing session, so if the application calls `entdel` a second time *during that session* and specifies the same entity, the entity is undeleted (`handent` can be used to retrieve the names of deleted entities, as previously described).

Note: Attributes and Polyline vertices cannot be deleted independently of their parent entities; `entdel` operates only on main entities. If you need to delete an attribute or vertex, you can use `command` to invoke the AutoCAD ATTEDIT or PEDIT commands.

The `entget` function returns the definition data of a specified entity. The data are returned as a list. Each item in the list is specified by a DXF group code. The first item in the list contains the entity's current name.

Example

For the following example, assume that the following (default) conditions apply to the current drawing:

- The current layer is 0.
- The current linetype is CONTINUOUS.
- The current elevation is zero.
- Entity handles are disabled.

Suppose the user has drawn a line with the following sequence of commands.

```
Command: Line
From point: 1,2
To point: 6,6
To point: ↵
```

Then an AutoLISP application could retrieve and print the definition data for the line by using the following AutoLISP function:

```
(defun C:PRINTDXF (/ ent entl ct)
   (setq ent (entlast))              Sets ent to the last entity
   (setq entl (entget ent))          Sets entl to the association list of the last entity
   (setq ct 0)                       Sets ct (a counter) to 0
   (textpage)                        Switches to the text screen
   (princ "\nResults from entget of last entity: ")
   (repeat (length entl)             Repeats for the number of members in the list
      (print (nth ct entl))          Prints a newline then each list member
      (setq ct (1+ ct))              Increments the counter by one
   )
   (princ)                           Exits quietly
)
```

This would print the following (the entity name value will vary):

```
Results from entget of last entity:
(–1 . <Entity name: 60000014>)
(0 . "LINE")
(8 . "0")
(10 1.0 2.0 0.0)
(11 6.0 6.0 0.0)
(210 0.0 0.0 1.0)
```

The first member of the list (with a –1 group code) contains the name of the entity this list represents. The **entmod** function described below uses it to identify the entity to be modified.

The codes for the components of the entity are those used by DXF and documented in appendix B of this manual and chapter 11 of the *AutoCAD Customization Manual*. As with DXF, the entity header items (color, linetype, thickness, the attributes-follow flag, and the entity handle) are returned only if they have values other than the default. Unlike DXF, optional entity definition fields are returned whether they equal their defaults, or not. This is intended to simplify processing: an application can always assume that these fields are present. Also unlike DXF, associated *X*, *Y*, and *Z* coordinates are returned as a single point variable rather than as separate *X* (10), *Y* (20), and *Z* (30) groups.

The **assoc** function searches a list for a group of a specified type:

```
(cdr (assoc 0 entl))
```

This would return the entity type "`LINE`" (group code 0) in the list `entl`. If the DXF group code specified is not present in the list (or if it is not a valid DXF group), **assoc** returns `nil`.

The **entmod** function modifies an entity. It passes a list that has the same format as a list returned by **entget**, but with some of the entity group values (presumably) modified by the application. This function complements **entget**: the primary mechanism by which an AutoLISP application updates the database is by retrieving an entity with **entget**, modifying its entity list, then passing the list back to the database via **entmod**.

Example

This code fragment retrieves the definition data of the first entity in the drawing and changes its layer property to `MYLAYER`:

```
(setq en (entnext))              Sets en to the name of the first entity in the drawing
(setq ed (entget en))            Sets ed to the entity data for entity name en
(setq ed
    (subst (cons 8 "MYLAYER")
      (assoc 8 ed)               Changes the layer group in ed
      ed                         to layer MYLAYER
    )
)
(entmod ed)                      Modifies entity en's layer in drawing
```

The following are the restrictions on the database changes that **entmod** is allowed to make:

- **entmod** cannot change the entity's type or handle.
- AutoCAD must recognize all objects (except layers) that the entity list refers to.

  The name of any Text style, Linetype, Shape, or Block that appears in an entity list must be defined in the current drawing *before* the entity list is passed to **entmod**.

  Exception: **entmod** accepts new layer names.

  If the entity list refers to a layer name that has not been defined in the current drawing, **entmod** creates a new layer. The attributes of the new layer are the standard default values used by the New option of the AutoCAD LAYER command.
- **entmod** cannot change internal fields (internal fields are the values that AutoCAD assigns to certain group codes: –2, entity name reference; –1, entity name; 5, entity handle).

  Any attempt to change an internal field—for example, the main entity name in a Seqend subentity (group –2)—is simply ignored.
- **entmod** cannot change Viewport entities.

  An attempt to change a Viewport entity causes an error.

The **entmod** function *can* modify subentities such as Polyline Vertices and Block Attributes.

Caution: If you use **entmod** to modify an entity in a Block definition, this affects all INSERT or XREF references to that Block; also, entities in Block definitions cannot be deleted by **entdel**.

An application can also add an entity to the drawing database by calling the **entmake** function. Like **entmod**, the argument to **entmake** is a list whose format is similar to that returned by **entget**. The new entity that the list describes is appended to the drawing database (it becomes the Last entity in the drawing). If the entity is a complex entity (a Polyline or Block), it is not appended to the database until it is complete, as described later in this section.

Example

The following code fragment creates a circle on the layer MYLAYER:

```
(entmake '( (0 . "CIRCLE")               Entity type
            (8 . "MYLAYER")              Layer
            (10 5.0 7.0 0.0)             Center point
            (40 . 1.0)                   Radius
          )
)
```

The restrictions on **entmake** are similar to those for **entmod**:

- The first or second member in the list *must* specify the entity type.

  The type must be a valid DXF group code.

If the first member does not specify the type, it can *only* specify the name of the entity: group –1 (the name is not saved in the database).

- AutoCAD must recognize all objects (except layers) that the entity list refers to.

  Exception: `entmake` accepts new layer names.

- Any internal fields passed to `entmake` are ignored.
- `entmake` cannot create Viewport entities.

Both `entmod` and `entmake` perform the same consistency checks on the entity data passed to them as the AutoCAD DXFIN command performs when reading DXF files. They will fail if they cannot create valid drawing entities.

Anonymous Blocks

The Block Definitions (BLOCK) table in a drawing can contain anonymous blocks, which AutoCAD creates to support hatch patterns and associative dimensioning. They can also be created by `entmake`, usually to contain entities that the user cannot access directly. Unreferenced anonymous blocks are *purged* from the BLOCK table at the beginning of each drawing session. Referenced (INSERTed) anonymous blocks are *not* purged. You can use `entmake` to create a block reference (INSERT) to an anonymous block (you cannot pass an anonymous block to the INSERT command). You can also use `entmake` to redefine the block. The entities in a block (but not the Block entity itself) can be modified with `entmod`.

The name (group 2) of an anonymous block created by AutoLISP or ADS has the form `*U`*nnn*, where *nnn* is a number generated by AutoCAD. Also, the low-order bit of an anonymous block's *Block type flag* (group 70) is set to one. When `entmake` creates a block whose name begins with `*` and whose anonymous bit is set, AutoCAD treats this as an anonymous block and assigns it a name. Any characters following the `*` in the name string passed to `entmake` are *ignored.*

Caution: Although a referenced anonymous block becomes permanent, the numeric portion of its name can change between drawing sessions. Applications cannot rely on anonymous block names remaining constant.

Creating Complex Entities

A complex entity (a Polyline or Block) is created by multiple calls to `entmake`, by using a separate call for each subentity. When `entmake` first receives an initial component for a complex entity, it creates a temporary file in which to gather the definition data (and extended data, if that is present; see the section "Notes on Extended Entity Data" on page 63). For each subsequent `entmake` call, the function checks to see if the temporary file exists. If it does, the new subentity is appended to the file. When the definition of the complex entity is complete (i.e., when `entmake` receives an appropriate Seqend or Endblk subentity), the entity is checked for consistency, and if it is valid it is added to the drawing. The file is deleted when the complex entity is complete or when its creation has been cancelled.

Caution: The entity does not appear in the drawing database until the final Seqend or Endblk subentity has been passed to `entmake`. In particular, `entlast` cannot be used to retrieve the most recently created subentity for a complex entity that has not been completed.

As the previous paragraphs imply, **entmake** can construct only one complex entity at a time. If a complex entity is being created and **entmake** receives invalid data or an entity that is not an appropriate subentity, both the invalid entity and the entire complex entity are rejected. You can explicitly cancel the creation of a complex entity by calling **entmake** with no arguments.

The following example contains five **entmake** functions that create a single complex entity, a Polyline. The Polyline has a linetype of DASHED and a colour of BLUE. It has three vertices located at coordinates (1,1,0), (4,6,0), and (3,2,0). All other optional definition data assume default values (for this example to work properly, the linetype DASHED must be loaded).

```
(entmake   '( (0 . "POLYLINE")           Entity type
              (62 . 5)                   Color
              (6 . "dashed")             Linetype
              (66 . 1)                   Vertices follow

)
(entmake   '( (0 . "VERTEX")             Entity type
              (10 1.0 1.0 0.0)           Start point
            )
)
(entmake   '( (0 . "VERTEX")             Entity type
              (10 4.0 6.0 0.0)           Second point
            )
)
(entmake   '( (0 . "VERTEX")             Entity type
              (10 3.0 2.0 0.0)           Third point
            )
)
(entmake   '((0 . "SEQEND")))            Sequence end
```

Block definitions begin with a Block entity and end with an Endblk subentity. Block definitions cannot be nested, nor can they reference themselves. A block definition *can* contain references to other block definitions.

Caution: The **entmake** function does not check for name conflicts in the Block Definitions table, so it can *redefine* existing blocks. Before you use it to create a block, you should use **tblsearch** (described in the section "Symbol Table Access" on page 69) to ensure that the name of the new block is unique.

Block Insert references can include an *attributes follow* flag (group 66). If present and equal to one, a series of Attribute (Attrib) entities is expected to follow the Insert entity. The attribute sequence is terminated by a Seqend subentity.

Polyline entities always include a *vertices follow* flag (also group 66); the value of this flag must be one, and the flag must be followed by a sequence of Vertex entities, terminated by a Seqend subentity.

Complex entities can exist in either model space or in paper space, but not both. If the current space is changed by invoking either MSPACE or PSPACE (via **command**) while a complex entity is under construction, a subsequent call to **entmake** will cancel the complex entity. This can also occur if the subentity has a 67 group whose value does not match the 67 group of the entity header.

Entity Data Functions and the Graphics Screen

Changes to the drawing made by the entity data functions are reflected on the graphics screen, provided the entity being deleted, undeleted, modified, or made is in an area and on a layer that is currently visible. There is one exception to this: when **entmod** modifies a subentity, it does not update the image of the entire (complex) entity. The reason for this should be clear: if, for example, an application were to modify 100 vertices of a complex Polyline with 100 calls to **entmod**, the time required to recalculate and redisplay the entire Polyline as each vertex was changed would be unacceptably slow. Instead, an application can perform a series of subentity modifications, then redisplay the entire entity at once with a single call to the **entupd** function.

Example

Suppose the first entity in the current drawing is a Polyline with several Vertices. The following code modifies the second Vertex of the Polyline, and then regenerates its screen image:

```
(setq e1 (entnext))                      Sets e1 to the Polyline's entity name
(setq v1 (entnext e1))                   Sets v1 to its first vertex
(setq v2 (entnext v1))                   Sets v2 to its second vertex
(setq v2d (entget v2))                   Sets v2d to the vertex data
(setq v2d
   (subst '(10 1.0 2.0 0.0)
      (assoc 10 v2d)                     Changes the vertex's location in v2d
      v2d                                to point (1,2,0)
   )
)
(entmod v2d)                             Moves the vertex in the drawing
(entupd e1)                              Regenerates the Polyline entity e1
```

The argument to **entupd** can specify either a main entity or a subentity: in either case, **entupd** regenerates the *entire* entity. Although its primary use is for complex entities, as shown in the example, **entupd** can regenerate any entity in the current drawing.

Caution: If the modified entity is in a block definition, then **entupd** is not sufficient: you must regenerate the drawing by invoking the AutoCAD REGEN command (via **command**) to ensure that all instances of the block references are updated.

Notes on Processing Curve-Fit and Spline-Fit Polylines

When an AutoLISP application uses **entnext** to step through the vertices of a Polyline, it might encounter vertices that were not created explicitly; auxiliary vertices are inserted automatically by the AutoCAD PEDIT command's Fit and Spline options. You can safely ignore them, since changes to these vertices will be discarded the next time the user uses PEDIT to Fit or Spline the Polyline.

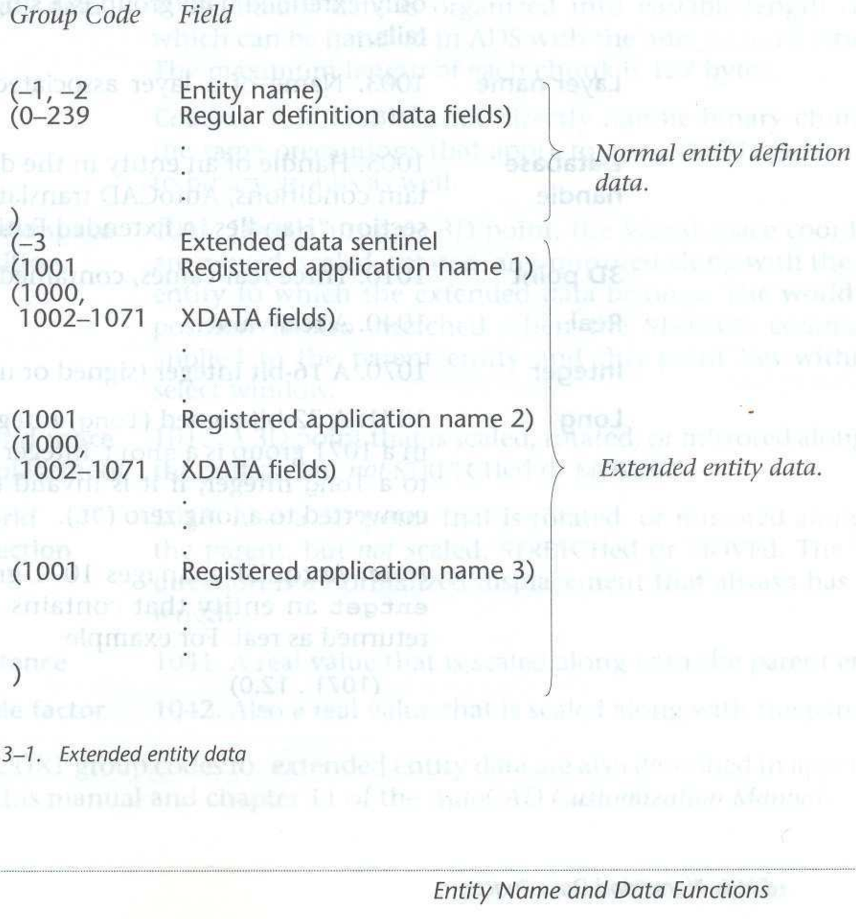
The Polyline entity's group 70 flags indicate whether the Polyline has been curve-fit (bit value 2) or spline-fit (bit value 4). If neither of these bits is set, all of the Polyline's vertices are regular user defined vertices. However, if the curve-fit bit (2) is set, alternating vertices of the Polyline will have bit value 1 set in their 70 group to indicate that they were inserted by the curve-fitting process. If you are using **entmod** to move the vertices of such a Polyline with the intent of refitting the curve using PEDIT, you should ignore these vertices. Likewise, if the Polyline entity's spline-fit flag bit (bit 4) is set, an assortment of vertices will be found, some with flag bit 1 (inserted by curve-fitting if system variable SPLINESEGS was negative), some with bit value 8 (inserted by spline fitting), and all others with bit value 16 (spline frame control point). Here again, if you are using **entmod** to move the vertices and intend to refit the spline afterward, you should move only the control point vertices.

Notes on Extended Entity Data

Several AutoLISP functions are provided to handle *extended entity data,* which is created by applications written with ADS or AutoLISP. Extended entity data was introduced in AutoCAD Release 11 for the use of both user applications and products such as the Application Programming Interface (API) of the Advanced Modelling Extension (AME). If an entity contains extended data, it follows the entity's regular, definition data. This is illustrated by figure 3-1.

An entity's extended data can be retrieved by calling **entget**. The **entget** function retrieves an entity's regular definition data and the extended data for those applications specified in the **entget** call.

Note: When extended entity data is retrieved via **entget**, the beginning of extended data is indicated by a –3 code; the –3 code is in a list that precedes the first 1001 group. The 1001 group contains the application name of the first application retrieved, as shown in the figure and described in the following sections.



```
Group Code     Field

(–1, –2        Entity name)
(0–239         Regular definition data fields)
   .
   .
)
(–3            Extended data sentinel
(1001          Registered application name 1)
(1000,
 1002–1071     XDATA fields)
   .
   .
(1001          Registered application name 2)
(1000,
 1002–1071     XDATA fields)
   .
   .
(1001          Registered application name 3)
   .
   .
)
```

Figure 3–1. Extended entity data

Organization of Extended Entity Data

As you can observe in the figure, extended data consists of one or more 1001 groups, each of which begins with a unique *application name*. Application names are string values.

The extended data groups returned by `entget` follow the definition data in the order they're saved in the database. This is also illustrated schematically in the figure.

Within each application's group, the contents, meaning, and organization of the data are defined by the application itself; AutoCAD maintains the information, but doesn't use it. As the figure also shows, the group codes for extended entity data are in the range 1000–1071. Many of these group codes are for familiar data types, as follows:

String 1000. Strings in extended entity data can be up to 255 bytes long (with the 256th byte reserved for the null character).

Application name 1001 (also a string value). Application names can be up to 31 bytes long (the 32d byte is reserved for the null character) and must adhere to the rules for symbol table names (such as layer names). An application name can contain letters, digits, and the special characters $ (dollar sign), - (hyphen), and _ (underscore). It *cannot* contain spaces. Letters in the name are converted to uppercase. The use of application names is described in more detail later in this section.

Note: A group of extended data *cannot* consist of an application name with no other data. If you attempt to add a 1001 group but no other extended data to an existing entity, the attempt is ignored. If you attempt to make an entity whose only extended data group is a single 1001 group, the attempt fails.

Layer name 1003. Name of a layer associated with the extended entity data.

Database handle 1005. Handle of an entity in the drawing database. Under certain conditions, AutoCAD translates these, as described in the section "Handles in Extended Entity Data" on page 68.

3D point 1010. Three real values, contained in a point.

Real 1040. A real value.

Integer 1070. A 16-bit integer (signed or unsigned).

Long 1071. A 32-bit signed (`long`) integer. If the value that appears in a 1071 group is a `short` integer or real value, it is converted to a `long` integer; if it is invalid (for example, a string), it is converted to a long zero (`0L`).

Note: AutoLISP manages 1071 groups as real values. If you `entget` an entity that contains a 1071 group, the value is returned as real. For example:

```
(1071 . 12.0)
```

If you want to *create* a 1071 group in an entity via **entmake** or **entmod**, you can use either a real or an integer value. For example:

```
(entmake '((.... (1071 . 12) .... )))
(entmake '((.... (1071 . 12.0) .... )))
(entmake '((.... (1071 . 65537.0) .... )))
(entmake '((.... (1071 . 65537) .... )))
```

but AutoLISP still returns the group value as a real:

```
(entmake '((.... (1071 . 65537) .... )))
```

returns

(1071 . 65537.0)

ADS always manages 1071 groups as `long` integers.

Several other extended entity data groups have special meaning in this context (if the application chooses to use them):

Control string 1002. An extended data control string can be either `"{"` or `"}"`; these braces enable the application to organize its data by subdividing it into lists. The left-brace begins a list, and a right-brace terminates the most recent list; lists can be nested.

When it reads the extended entity data for a particular application, AutoCAD checks to ensure that braces are balanced correctly.

Caution: If a 1001 group appears within a list, it is simply treated as a string and *does not begin* a new application group.

Binary data 1004. Binary data is organized into variable-length *chunks,* which can be handled in ADS with the `ads_binary` structure. The maximum length of each chunk is 127 bytes.

Caution: AutoLISP cannot directly handle binary chunks, so the same precautions that apply to `long` (1071) groups apply to binary groups as well.

World space position 1011. Unlike a simple 3D point, the World space coordinates are moved, scaled, rotated, and mirrored along with the *parent* entity to which the extended data belongs. The world space position is also stretched when the STRETCH command is applied to the parent entity and this point lies within the select window.

World space displacement 1012. A 3D point that is scaled, rotated, or mirrored along with the parent, but *not* STRETCHed or MOVEd.

World direction 1013. Also a 3D point that is rotated, or mirrored along with the parent, but *not* scaled, STRETCHed or MOVEd. The World direction is a normalized displacement that always has a unit length.

Distance 1041. A real value that is scaled along with the parent entity.

Scale factor 1042. Also a real value that is scaled along with the parent.

The DXF group codes for extended entity data are also described in appendix B of this manual and chapter 11 of the *AutoCAD Customization Manual.*

Registering an Application

Application names are saved not only with the extended data of each entity that uses them, but also in the APPID table. An application must *register* the name or names that it uses. In AutoLISP, this is done by a call to **regapp**. The **regapp** function simply specifies a string to use as an application name. If it successfully adds the name to APPID, it returns the name of the application, and `nil` if it can't. A result of `nil` usually indicates that the name is already present in the symbol table; this is not an actual error condition, but a normally expected return value, since the application name only needs to be registered once per drawing.

To register itself, an application should first check that its name is not already in the APPID table, since **regapp** only needs to be called once per drawing. If the name is not there, the application must register it; otherwise, it can simply go ahead and use the data, as described in the following subsection.

Example

The following fragment shows the typical use of **regapp** (the **tblsearch** function is described in the section "Symbol Table Access" on page 69):

```
(setq appname "MYAPP_2356")                          Unique application name
(if (tblsearch "appid" appname)                      Checks if it is already registered
  (princ (strcat "\n" appname " already registered. "))
  (if (= (regapp appname) nil)                        Some other problem
    (princ (strcat "\nCan't register XDATA for " appname ". "))
  )
)
```

Important: The **regapp** function provides a measure of security, but it cannot guarantee that two different applications have not chosen the same 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).

Retrieving Extended Entity Data

An application can obtain the extended entity data it has registered by calling **entget**. The **entget** function can return both the definition data and the extended data for the applications it requests; it requires an additional argument, *`application`*, that specifies the application names. The names passed to **entget** must correspond to applications that have been registered by a previous call to **regapp**; they can also contain wild card characters.

Examples

By default Hatch patterns contain extended entity data. The following code allows you to see the association list of this extended entity data. Entering this code at the command line

Command: **(entget (car (entsel)) '("ACAD"))** *Select a Hatch entity*

should return a list that looks something like this:

```
((-1 . <Entity name: 600000c0>) (0 . "INSERT") (8 . "0") (2 . "*X0") (10 0.0 0.0 0.0)
(41 . 1.0) (42 . 1.0) (50 . 0.0) (43 . 1.0) (70 . 0) (71 . 0) (44 . 0.0) (45 . 0.0)
(210 0.0 0.0 1.0) (-3 ("ACAD" (1000 . "HATCH") (1002 . "{") (1070 . 16)
(1000 . "LINE") (1040 . 1.0) (1040 . 0.0) (1002 . "}"))))
```

This fragment shows a typical sequence for retrieving extended entity data for two specified applications. Note that the `application` argument passes application names in list form.

```
(setq working_elist (entget ent_name
  '(("MY_APP_1")("SOMETHING_ELSE"))    Only extended data from "MY_APP_1"
)                                      and "SOMETHING_ELSE" is retrieved
(if working_elist
  (progn
    .                                  Updates working entity groups
    .
    .
   (entmod working_elist)              Only extended data from registered applications
  )                                    still in the working_elist list are modified
)
```

As the sample code shows, extended entity data retrieved by **entget** can be modified by a subsequent call to **entmod**, just as **entmod** is used to modify normal definition data (extended entity data can also be created by defining it in the entity list passed to **entmake**).

Returning the extended data of only those applications specifically requested protects one application from messing up another's data. It also controls the amount of memory an application needs to use, and simplifies the extended data processing an application needs to perform.

Caution: Since the strings passed via `application` can include wild card characters, an application name of `"*"` will cause **entget** to return *all* extended data attached to an entity.

Attaching Extended Entity Data to an Entity

You can use extended entity data to store almost any type of information you want; its use is limited only by your imagination. Following is a simple example of attaching extended entity data to an entity.

Example

You must first draw a simple entity (e.g., Line or Circle), and then enter the following code:

```
(setq lastent (entget (entlast))      Gets the association list of definition data for
                                      the last entity
(regapp "NEWDATA")                    Registers the application name
(setq exdata                          Sets the variable exdata equal to the new
  '((-3 ("NEWDATA"                    extended entity data,
    (1000 . "This is a new thing!"))) in this case, a text string
 )
)
(setq newent (append lastent exdata)) Appends the new data list to the entity's list
(entmod newent)                       Modifies the entity with the new
                                      definition data
```

To verify that your new extended entity data has been attached to the entity, enter the following code and select the entity:

```
(entget (car (entsel)) '("NEWDATA"))
```

This example might not be the most practical use for extended entity data; it does, however, show the basic method for attaching extended entity data to an entity.

See also: The *xdata.lsp* routine in the *sample/* directory.

Managing Extended Entity Data Memory Use

Extended entity data is currently limited to 16K per entity. Since the extended data of an entity can be created and maintained by multiple applications, this can lead to problems when the size of the extended entity data approaches its limit. AutoLISP provides two functions, `xdsize` and `xdroom`, to assist in managing the memory extended entity data occupies. When `xdsize` is passed a list of extended entity data, it returns the amount of memory (in bytes) that data will occupy; when `xdroom` is passed the name of an entity, it returns the remaining number of *free* bytes that can still be appended to the entity.

Suggestion: The `xdsize` function must read an extended entity data list, which can be large. Because of this, this function can be slow, and it is not recommended that you call it frequently. A better approach is to use it (in conjunction with `xdroom`) in an error handler: if a call to `entmod` fails, you can use `xdsize` and `xdroom` to find out whether the call failed because the entity has run out of extended data, and take appropriate action if that's the reason it failed.

Handles in Extended Entity Data

Extended entity data can contain handles (group 1005) to save relational structures within a drawing. One entity can reference another by saving the other's handle in its extended data; the handle can later be retrieved from extended data and passed to `handent` to obtain the other entity. Since more than one entity can reference another, extended data handles are not necessarily unique; the AUDIT command does require that handles in extended data either be NULL or valid entity handles (within the current drawing). The best way to ensure extended entity handles are valid is to obtain a referenced entity's handle directly from its definition data, via `entget` (if handles are currently turned on, the handle value is in group 5).

Suggestion: To reference entities in other drawings (e.g., ones that are attached via XREF), you can avoid protests from AUDIT by using extended entity strings (group 1000) rather than handles (group 1005), since the handles of cross-referenced entities are either *not* valid in the current drawing, or conflict with valid handles. However, if an XREF Attach should change to an XREF Bind or be combined into the current drawing in some other way, it is up to the application to revise entity references accordingly.

Important: When drawings are combined via INSERT, INSERT*, XREF Bind (XBIND), or partial DXFIN, handles are translated so they become valid in the current drawing, as described in chapter 10 of the *AutoCAD Reference Manual* (if the incoming drawing did not employ handles, new ones are assigned). Extended entity handles that refer to incoming entities are *also translated* when these commands are invoked.

When an entity is placed in a Block definition (via the BLOCK command), the entity within the block is assigned new handles. (If the original entity is restored via OOPS, it retains its original handles.) The value of any extended data handles remains unchanged. When a Block is exploded (via the EXPLODE command), extended data handles *are* translated, in a manner similar to the way they are translated when drawings are combined. If the extended data handle refers to an entity not within the block, it is unchanged; but if the extended data handle refers to an entity within the block, it is assigned the value of the new (exploded) entity's handle.

Symbol Table Access

The **tblnext** function sequentially scans symbol table entries, and the **tblsearch** function retrieves specific entries. Table names are specified by strings. The valid names are: "LAYER", "LTYPE", "VIEW", "STYLE", "BLOCK", "UCS", "VPORT", and "APPID". Both these functions return lists with DXF group codes, much like the entity data returned by **entget**.

The first call to **tblnext** returns the first entry in the specified table. Subsequent calls that specify the same table return successive entries, unless the second argument to **tblnext** (*rewind*) is nonzero, in which case, **tblnext** returns the first entry again.

Example

The function **GETBLOCK** shown here retrieves the symbol table entry for the first block (if any) in the current drawing, then displays it in a list format.

```
(defun C:GETBLOCK (/ blk ct)
  (setq blk (tblnext "BLOCK" 1))         Gets the first BLOCK entry
  (setq ct 0)                            Sets ct (a counter) to 0
  (textpage)                             Switches to the text screen
  (princ "\nResults from GETBLOCK: ")
  (repeat (length blk)                   Repeats for the number of members in the list
     (print (nth ct blk))                Prints a newline then each list member
     (setq ct (1+ ct))                   Increments the counter by one
  )
  (princ)                                Exits quietly
)
```

Entries retrieved from the BLOCK table contain a –2 group that contains the name of the first entity in the block definition. If the block is empty, this is the name of the block's ENDBLK entity, which is never seen on non-empty blocks. Thus, given a drawing with a single block named BOX, a call to **GETBLOCK** would print the following (the name value will vary from session to session):

```
Results from GETBLOCK:
(0 . "BLOCK")
(2 . "BOX")
(70 . 0)
(10 9.0 2.0 0.0)
(–2 . <Entity name: 40000126>)
```

As with **tblnext**, the first argument to **tblsearch** is a string that names a table, but the second argument is a string that names a particular symbol in the table. If the symbol is found, **tblsearch** returns its data. This function has a third argument, *setnext*, that can be used to coordinate operations with **tblnext**. If *setnext* is nil, the **tblsearch** call has no effect on **tblnext**, but if *setnext* is non-nil, the next call to **tblnext** returns the table entry following the entry found by **tblsearch**.

The *setnext* option is especially useful when dealing with the VPORT symbol table, because all viewports in a particular viewport configuration have the same name (e.g., *ACTIVE).

Reminder: If the VPORT symbol table is accessed when TILEMODE is Off, any changes will have no visible effect until TILEMODE is set back On (TILEMODE is set either by the SETVAR command or by entering its name directly, as described in the *AutoCAD Reference Manual*). Don't confuse Vports described by the VPORT symbol table with paper-space Viewport entities.

Example

The following code will process each viewport in the configuration, 4VIEW:

```
(setq v (tblsearch "VPORT" "4VIEW" T))        Finds first VPORT entry
(while (and v (= (cdr (assoc 2 v)) "4VIEW"))
         .                                    Processes entry. . .
         .
         .
   (setq v (tblnext "VPORT"))                 Gets next VPORT entry
)
```

Chapter 4
AutoLISP Functions

This chapter describes all of the functions provided by AutoLISP. It consists of a synopsis and a catalogue of the functions. In the synopsis, function names are grouped by topic, and each is followed by a brief description. In the catalogue, function names appear in alphabetical order, and the functions are described in detail.

Synopsis of Functions

AutoLISP Functions Common to ADS

Function Handling

```
(defun sym argument-list expr ...)
```

Defines an external function (Subr).

Error Handling

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

Prints an error message.

```
(alert string)
```

Displays a dialogue box alerting the user with *string*.

AutoCAD Queries and Commands

```
(command [arguments] ...)
```

Executes one or more AutoCAD commands.

```
(getvar varname)
```

Gets the current value of an AutoCAD system variable.

```
(setvar varname value)
```

Sets the value of an AutoCAD system variable.

(findfile *filename***)**

Searches for a filename.

(getfiled *title filename ext flags***)**

Prompts the user for a filename via the standard AutoCAD file dialogue box.

(osnap *pt mode-string***)**

Finds a point via object snap.

Geometric Utilities

(distance *pt1 pt2***)**

Finds the distance between two points.

(angle *pt1 pt2***)**

Finds the angle between two lines.

(polar *pt angle dist***)**

Finds a point via polar coordinates.

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

Finds the intersection of two lines.

(textbox *elist***)**

Returns the diagonal coordinates of a box that encloses a text entity.

User Input

(initget *[bits][string]***)**

Determines valid user input for the next call to a **getxxx** function.

(getreal *[prompt]***)**

Prompts for user input of a real (floating-point) number.

(getstring *[cr][prompt]***)**

Prompts for user input of a string.

(getpoint *[pt][prompt]***)**

Prompts for user input of a point.

(getcorner *pt [prompt]***)**

Prompts for user input of the corner of a rectangle.

`(getdist [pt][prompt])`

Prompts for user input of a distance.

`(getangle [pt][prompt])`

Prompts for user input of an angle.

`(getorient [pt][prompt])`

Similar to `getangle`, but takes into account the current value of the ANGBASE system variable.

`(getkword [prompt])`

Prompts for user input of a keyword.

`(getint [prompt])`

Prompts for user input of an integer.

Conversion

`(rtos number [mode [precision]])`

Formats a real (floating-point) value as a string.

`(distof string [mode])`

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

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

Formats an angle as a string.

`(angtof string [mode])`

Converts a string that displays an angle into a real (floating-point) value.

`(cvunit value from to)`

Converts between real-world units.

Coordinate System Transformation

`(trans pt from to [disp])`

Translates a point or displacement from one coordinate system to another.

Digitizer Calibration

`(tablet code [row1 row2 row3 direction])`

Controls digitizer calibration.

Display Control

(prin1 *[expr [file-desc]]***)**

Prints a message on the text screen or to an open file.

(princ *[expr [file-desc]]***)**

Prints a message on the text screen or to an open file.

(print *[expr [file-desc]]***)**

Prints a message on the text screen or to an open file.

(prompt *msg***)**

Displays a message on the prompt line.

(menucmd *string***)**

Displays and activates menus.

(redraw *[ename [mode]]***)**

Redraws the current graphics screen.

(graphscr)

Displays the current graphics screen.

(textscr)

Displays the current text screen.

(textpage)

Same as **textscr**, but clears the text screen first.

Low-level Graphics

(grclear)

Clears the graphics screen.

(grdraw *from to color [highlight]***)**

Draws a vector in the current viewport.

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

Draws multiple vectors in the current viewport.

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

Reads from an input device.

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

Displays text in the menu, mode, or status area of the graphics screen.

Wild Card Matching

(wcmatch *string pattern***)**

Matches a string to a wild card pattern.

Selection Sets

(ssget *[mode][pt1 [pt2]][pt-list][filter-list]***)**

Gets a selection set.

(ssadd *[ename [ss]]***)**

Adds an entity to a selection set (or creates a new set).

(ssdel *ename ss***)**

Deletes an entity from a selection set.

(sslength *ss***)**

Returns the number of entities in a selection set.

(ssname *ss index***)**

Returns the name of an entity in a selection set.

(ssmemb *ename ss***)**

Checks whether an entity is a member of a selection set.

Entity Handling

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

Gets the definition data of an entity.

(entmod *elist***)**

Modifies the definition data of an entity.

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

Makes a new entity and appends it to the drawing database.

(entdel *ename***)**

Deletes (and undeletes) entities in the drawing.

(entnext *[ename]***)**

Finds the next entity in the drawing.

(entlast)

Finds the last entity in the drawing.

(handent *handle***)**

Finds an entity by its handle.

(entsel *[prompt]***)**

Prompts user to select an entity by specifying a point.

(nentsel *[prompt]***)**

Like **entsel**, but returns additional data for nested entities.

(nentselp *[prompt][pt]***)**

Similar to **nentsel** but returns a full 3D 4×4 matrix and enables the program to specify the pick point.

(entupd *ename***)**

Updates the screen image of an entity.

Extended Entity Data

(regapp *application***)**

Registers the application's extended entity data.

(xdsize *list***)**

Returns the amount of memory (in bytes) that a list of extended entity data will occupy.

(xdroom *ename***)**

Returns the amount of memory (in bytes) that an entity has available for extended data.

Symbol Tables

(tblnext *table-name [rewind]***)**

Finds the next item in a symbol table.

(tblsearch *table-name symbol [setnext]***)**

Searches for a symbol in a symbol table.

General Functions

Arithmetic

(+ *number number ...***)**

Returns the sum of all numbers.

(- *number [number ...]***)**

Subtracts the second number from the first and returns the difference.

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

Returns the product of all numbers.

`(/ number [number ...])`

Divides the first number by the second and returns the quotient.

`(~ number)`

Returns the bitwise NOT of *number*.

`(1+ number)`

Returns *number* incremented by 1.

`(1- number)`

Returns *number* decremented by 1.

`(abs number)`

Returns the absolute value of *number*.

`(atan num1 [num2])`

Returns the arctangent of a number in radians.

`(cos angle)`

Returns the cosine of an angle.

`(exp number)`

Returns a value raised to the *number* power (natural antilog).

`(expt base power)`

Returns *base* raised to *power*.

`(fix number)`

Returns the conversion of a number into an integer.

`(float number)`

Returns the conversion of a number into a real value.

`(gcd num1 num2)`

Returns the greatest common denominator of two numbers.

`(log number)`

Returns the natural log of a number as a real value.

`(logand number number ...)`

Returns the result of a logical bitwise AND of a list of numbers.

`(logior integer ...)`

Returns the result of a logical bitwise inclusive OR of a list of numbers.

(**lsh** *num1 numbits*)

Returns the logical bitwise shift of a number by a given number of bits.

(**max** *number number ...*)

Returns the largest of the numbers given.

(**min** *number number ...*)

Returns the smallest of the numbers given.

(**minusp** *item*)

Verifies that *item* is a real or integer and evaluates to a negative value.

pi

Evaluates to constant π.

(**rem** *num1 num2 ...*)

Divides two numbers and returns the remainder.

(**sin** *angle*)

Returns the sine of an angle as a real value.

(**sqrt** *number*)

Returns the square root of a number as a real value.

(**zerop** *item*)

Verifies that *item* is a real number or an integer that evaluates to zero.

Symbol Handling

(**atom** *item*)

Verifies that *item* is an atom.

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

Returns a list of previously defined functions.

(**boundp** *atom*)

Verifies that a value has been bound to an atom.

(**not** *item*)

Verifies that *item* is `nil`.

(**null** *item*)

Verifies that *item* is bound to `nil`.

`(numberp item)`

Verifies that *item* is a real or an integer.

`(quote expr ...)`

Returns an expression unevaluated.

`(set sym expr)`

Sets the value of a quoted symbol to that of an expression.

`(setq sym1 expr1 [sym2 expr2] ...)`

Sets the value of one or more symbols to that of an expression.

`(type item)`

Returns the type of *item*.

Text Strings

`(read string)`

Returns the first list or atom obtained from the string.

`(read-char [file-desc])`

Reads a single character from the keyboard or from an open file.

`(read-line [file-desc])`

Reads a string from the keyboard or from an open file.

`(strcase string [which])`

Returns a copy of a string with all characters converted to upper or lowercase.

`(strcat string1 [string2] ...)`

Returns the concatenation of one or more strings.

`(strlen [string] ...)`

Returns the length, in characters, of a string.

`(substr string start [length])`

Returns a substring of a string.

`(write-char num [file-desc])`

Writes one character, described by an ASCII code, to the screen or an open file.

`(write-line string [file-desc])`

Writes a string to the screen or to an open file.

Conversion

`(ascii string)`

Returns the conversion of the first character of a string into its ASCII character code.

`(atof string)`

Returns the conversion of a string into a real value.

`(atoi string)`

Returns the conversion of a string into an integer.

`(chr integer)`

Returns the conversion of an integer representing an ASCII character code into a single-character string.

`(itoa int)`

Returns the conversion of an integer into a string.

Equality/Conditional

`(= atom atom ...)`

The *equal to* relational function.

`(/= atom atom ...)`

The *not equal to* relational function.

`(< atom atom ...)`

The *less than* relational function.

`(<= atom atom ...)`

The *less than or equal to* relational function.

`(> atom atom ...)`

The *greater than* relational function.

`(>= atom atom ...)`

The *greater than or equal to* relational function.

`(and expr ...)`

Returns the logical AND of a list of expressions.

`(Boole func int1 int2 ...)`

A general bitwise Boolean function.

`(cond (test1 result1 ...) ...)`

Primary conditional function in AutoLISP.

`(eq expr1 expr2)`

Determines whether two expressions are identical.

`(equal expr1 expr2 [fuzz])`

Determines whether two expressions evaluate to the same thing.

`(if testexpr thenexpr [elseexpr])`

Conditionally evaluates expressions.

`(or expr ...)`

Returns the logical OR of a list of expressions.

`(repeat number expr ...)`

Evaluates each expression a given number of times.

`(while testexpr expr ...)`

Repeats the enclosed expressions while the test expression remains true.

List Manipulation

`(append expr ...)`

Takes any number of lists and runs them together as one list.

`(assoc item alist)`

Searches an association list using `item` as a key, and returns the associated entry.

`(car list)`

Returns the first element of a list.

`(cdr list)`

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

`(caar list)`, `(cadr list)`, `(cddr list)`, `(cadar list)`, etc.

Concatenations up to four levels deep are supported.

`(cons new-first-element list)`

Returns a list with the new element added to the beginning.

`(foreach name list expr ...)`

Steps through a list and evaluates each expression for every element in the list.

`(list expr ...)`

Creates a list from any number of expressions.

(listp *item***)**

Verifies that *item* is a list.

(mapcar *function list1 ... listn***)**

Returns a list as the result of executing a function with the elements of lists supplied.

(member *expr list***)**

Searches a list for an occurrence of an expression and returns the remainder of the list starting with the first occurrence of the expression.

(nth *n list***)**

Returns the nth element of a list.

(reverse *list***)**

Returns a list with its elements reversed.

(subst *newitem olditem list***)**

Returns a copy of a list with *newitem* in place of every *olditem*.

File Handling

(close *file-desc***)**

Closes a file.

(load *filename [onfailure]***)**

Loads a file of AutoLISP expressions.

(open *filename mode***)**

Opens a file for access by the AutoLISP I/O functions.

ADS Application Handling

(ads)

Returns a list of the currently loaded AutoCAD Development System (ADS) applications.

(xload *application [onfailure]***)**

Loads an ADS application.

(xunload *application [onfailure]***)**

Unloads an ADS application.

Display

`(terpri)`

Prints a newline on the screen.

`(vports)`

Returns a list of viewport descriptors for the current viewport configuration.

Function Handling

(apply *function list***)**

Executes a function with the arguments given.

(eval *expr***)**

Returns the result of evaluating any AutoLISP expression.

(exit)

Forces the current application to quit.

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

Defines an anonymous function.

(progn *expr ...***)**

Evaluates each expression sequentially.

(trace *function ...***)**

Sets the trace flag for the specified functions.

(quit)

Forces the current application to quit.

(untrace *function ...***)**

Clears the trace flag for specified functions.

Memory Management

(alloc *number***)**

Sets the segment size to a given number of nodes.

(expand *number***)**

Allocates node space by requesting a specified number of segments.

(gc)

Forces a garbage collection.

(mem)

Displays the current state of AutoLISP's memory.

Miscellaneous

(getenv *variable-name***)**

Returns the string value assigned to a system environment variable.

(ver)

Returns a string containing the current AutoLISP version.

ADS Defined AutoLISP Functions

(acad_colordlg *colornum [flag]***)**

Displays the standard AutoCAD colour selection dialogue.

(acad_helpdlg *helpfile topic***)**

Displays the standard AutoCAD Help dialogue.

(acad_strlsort *list***)**

Sorts a list of strings.

ADS Defined Commands

(c:bhatch *pt [ss [vector]]***)**

Calls the BHATCH command and performs a boundary hatch on the specified area.

(c:bpoly *pt [ss [vector]]***)**

Calls the BPOLY command and creates a boundary Polyline.

(bherrs)

Gets an error message generated by a failed call to **c:bhatch** or **c:bpoly**.

(c:psdrag *mode***)**

Calls the PSDRAG command and sets the integer value *mode*.

(c:psfill *ent pattern arg1 [arg2] ...***)**

Fills a Polyline with a Postscript fill pattern.

(c:psin *filename position scale***)**

Inserts an encapsulated Postscript file.

Programmable Dialogue Box Functions

Detailed explanations of the following AutoLISP functions, which handle user-defined, customized dialogue boxes, are available in chapter 9 of the *AutoCAD Customization Manual*

This section summarizes the functions in the Programmable Dialogue Box (PDB) package, grouping them by functionality. These functions call an associated DCL (Dialogue Control Language) file to display the desired dialogue box. It shows the arguments to each function.

Opening and Closing DCL Files

`(`**`load_dialog`** *`filename`*`)`

Loads the specified DCL file.

`(`**`unload_dialog`** *`dcl_id`*`)`

Unloads the specified DCL file.

Opening and Closing Dialogue Boxes

`(`**`new_dialog`** *`dlgname dcl_id [[action-expression] screen-pt]`*`)`

Initializes a dialogue box and displays it.

`(`**`start_dialog`**`)`

Begins accepting user input from the dialogue box initialized by the **`new_dialog`** call.

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

Terminates the current dialogue box and stops displaying it. Must be called from within an action expression or callback function. This function also returns the current (*X*,*Y*) position of the dialogue box.

`(`**`term_dialog`**`)`

Terminates *all* current dialogue boxes as if the user had cancelled them.

Initializing Action Expressions or Callback Functions

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

Associates the specified tile with the action expression or callback function.

Handling Tiles and Attributes

`(`**`mode_tile`** *`key mode`*`)`

Sets the *`mode`* of the specified tile.

(get_attr *key attribute*)

Gets the DCL *value* of the specified attribute.

(get_tile *key*)

Gets the run-time value of the specified tile.

(set_tile *key value*)

Sets the run-time value of the specified tile.

Setting Up List Boxes and Popup Lists

(start_list *key [operation [index]]*)

Starts processing the specified list box or popup list.

(add_list *item*)

Adds the specified string to the current list.

(end_list)

Ends processing of the current list.

Creating Images

(dimx_tile *key*)

(dimy_tile *key*)

Retrieves dimensions of the specified tile.

(start_image *key*)

Starts creating the specified image.

(vector_image *x1 y1 x2 y2 color*)

Draws a vector in the currently active image.

(fill_image *x1 y1 x2 y2 color*)

Draws a filled rectangle in the currently active image.

(slide_image *x1 y1 x2 y2 sldname*)

Draws an AutoCAD slide in the currently active image.

(end_image)

Ends creation of the currently active image.

Application-specific Data

(client_data_tile *key clientdata*)

Associates application-managed data with the specified tile.

Catalogue of AutoLISP Functions

This section contains a description of the basic AutoLISP functions.

(+ *number number ...*)

This function returns the sum of all `numbers`. You can use it with real numbers or integers. If all the `numbers` are integers, the result is an integer; if any of the `numbers` are real numbers, the integers are promoted to real numbers and the result is a real number. For example:

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

(– *number [number] ...*)

This function subtracts the second `number` from the first and returns the difference. If more than two `numbers` are given, the sum of the second through last is subtracted from the first, and the final result is returned. If only one `number` is given, the result of subtracting it from zero is returned. You can use this function with reals or integers, with standard rules of promotion. For example:

`(- 50 40)`	returns	`10`
`(- 50 40.0 2)`	returns	`8.0`
`(- 50 40.0 2.5)`	returns	`7.5`
`(- 8)`	returns	`-8`

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

This function returns the product of all `numbers`. You can use this with reals or integers, with standard rules of promotion. If only one `number` is given, the result of multiplying it by 1 is returned. For example:

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

(/ number [number] ...)

This function divides the first *number* by the second and returns the quotient. If more than two *numbers* are given, it divides the first *number* by the product of the second through last, and returns the final quotient. You can use this with reals or integers, with standard rules of promotion. If only one *number* is given, the result of dividing it by 1 is returned. Examples follow:

`(/ 100 2)`	returns	`50`
`(/ 100 2.0)`	returns	`50.0`
`(/ 100 20.0 2)`	returns	`2.5`
`(/ 100 20 2)`	returns	`2`
`(/ 135 360)`	returns	`0`
`(/ 135 360.0)`	returns	`0.375`
`(/ 4)`	returns	`4`

(= atom atom ...)

This is the *equal to* relational function. It returns `T` if all the specified *atoms* are numerically equal, and `nil` otherwise. This function is valid for numbers and strings. These are 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`

Related topics: Compare this function with the **`eq`** and **`equal`** functions starting on page 112.

(/= atom atom ...)

This is the *not equal to* relational function. It returns `T` if *atom1* is not numerically equal to *atom2*, and `nil` if the two atoms are numerically equal. The function is undefined if more than two arguments are supplied. For example:

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

(< atom atom ...)

This is the *less than* relational function. It returns `T` if the first *atom* is numerically less than the second, and `nil` otherwise. If more than two *atoms* are given, it returns `T` if each atom is less than the *atom* to its right. For example:

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

(<= *atom atom ...*)

This is the *less than or equal to* relational function. It returns `T` if the first *`atom`* is numerically less than or equal to the second, and `nil` otherwise. If more than two *`atoms`* are given, it returns `T` if each atom is less than or equal to the *`atom`* to its right. For example:

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

(> *atom atom ...*)

This is the *greater than* relational function. It returns `T` if the first *`atom`* is numerically greater than the second, and `nil` otherwise. If more than two *`atoms`* are given, it returns `T` if each atom is greater than the *`atom`* to its right. For example:

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

(>= *atom atom ...*)

This is the *greater than or equal to* relational function. It returns `T` if the first *`atom`* is numerically greater than or equal to the second, and `nil` otherwise. If more than two *`atoms`* are given, it returns `T` if each atom is greater than or equal to the *`atom`* to its right. For example:

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

(~ *number*)

This function returns the bitwise NOT (one's complement) of *`number`*. The *`number`* argument must be an integer. For example:

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

(1+ *number*)

This function returns `number` increased by 1 (incremented). The `number` argument can be a real or an integer. For example:

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

(1– *number*)

This function returns `number` reduced by 1 (decremented). The `number` argument can be a real or an integer. For example:

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

(abs *number*)

This function returns the absolute value of `number`. The `number` argument can be a real or an integer. For example:

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

(ads)

Returns a list of the currently loaded AutoCAD Development System (ADS) applications. Each application and its path is a quoted string in the list. For example:

`(ads)`	might return	`("files/progs/PROG1" "PROG2")`

Related topics: See the **`xload`** and **`xunload`** functions starting on page 171.

(alert *string*)

Displays an alert box with the error or warning message passed in the `string` argument. An alert box is a dialogue box with a single OK button. For example:

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

You can display multiple lines 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 dependant. AutoCAD truncates any string too long to fit inside an alert box.

(alloc *number*)

Sets the segment size to a given *number* of nodes. See "Manual Allocation" on page 179 for more information on **alloc**.

(and *expr ...*)

This function returns the logical AND of a list of expressions. It ceases further evaluation and returns `nil` if any of the expressions evaluate to `nil`; otherwise it returns `T`. For example, given the following assignments:

```
(setq a 103) (setq b nil) (setq c "string")
```

then

```
(and 1.4 a c)          returns       T
(and 1.4 a b c)        returns       nil
```

(angle *pt1 pt2*)

This function returns the angle of a straight line running from UCS point *pt1* to UCS point *pt2*. 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. For example:

```
(angle '(1.0 1.0) '(1.0 4.0))        returns     1.5708
(angle '(5.0 1.33) '(2.4 1.33))      returns     3.14159
```

See also: "Geometric Utilities" on page 25.

(angtof *string [mode]*)

Converts *string*, which represents an angle in the display format specified by *mode*, into a floating-point value. The **angtof** function returns the result expressed in radians.

The *mode* argument specifies the units in which the string is formatted. The value should correspond to values allowed for the AutoCAD system variable AUNITS, as shown below. If *mode* is omitted, **angtof** uses the current value of AUNITS.

Table 4–1. Angular units values

Mode value	String format
0	Degrees
1	Degrees/minutes/seconds
2	Grads
3	Radians
4	Surveyor's units

The `string` must be a string that **angtof** can parse correctly according to the specified `mode`. It can be in the same form that **angtos** would return, or in a form that AutoCAD allows for keyboard entry. 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 `mode` values match).

If **angtof** succeeds, it returns a real value in radians; otherwise, it returns `nil`.

(angtos *angle [mode [precision]]*)

The **angtos** function takes `angle` (a real number, in radians) and returns it edited into a string according to the settings of `mode`, `precision`, the AutoCAD UNITMODE system variable, and the DIMZIN dimensioning variable. The `mode` and `precision` arguments are integers that specify the angular units mode and precision. The supported *mode* values are the same as those shown in table 4-1 (see the previous function description).

The `precision` argument is an integer that selects the number of decimal places of precision desired. The `mode` and `precision` correspond to the AutoCAD system variables AUNITS and AUPREC. If you omit these arguments, **angtos** uses the current settings of AUNITS and AUPREC, respectively.

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

```
(angtos 0.785398 0 4)          returns    "45.0000"
(angtos -0.785398 0 4)         returns    "315.0000"
```

The UNITMODE variable affects the returned string when surveyor's units are selected (a `mode` 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").

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: "String Conversions" on page 33.

(append *expr ...*)

This function takes any number of lists (`expr`) and runs them together as one list. For example:

```
(append '(a b) '(c d))           returns    (A B C D)
(append '((a)(b)) '((c)(d)))     returns    ((A)(B)(C)(D))
```

The **append** function requires that its arguments be lists.

(apply *function list*)

Executes the function specified by `function` with the arguments given by `list`. For example:

```
(apply '+ '(1 2 3))                   returns   6
(apply 'strcat '("a" "b" "c"))        returns   "abc"
```

The **apply** function works with both built-in functions (subrs) and user-defined functions (those created with either **defun** or **lambda**).

(ascii *string*)

This function returns the conversion of the first character of `string` into its ASCII character code (an integer). This is similar to the **ASC** function in the BASIC® language. For example:

```
(ascii "A")                           returns   65
(ascii "a")                           returns   97
(ascii "BIG")                         returns   66
```

(assoc *item alist*)

This function searches the association list `alist` for `item` as the key element and returns the `alist` entry. If `item` is not found as a key in `alist`, **assoc** returns `nil`. For example, assuming that list `al` is defined as

```
((name box) (width 3) (size 4.7263) (depth 5))
```

then

```
(assoc 'size al)                      returns   (SIZE 4.7263)
(assoc 'weight al)                    returns   nil
```

Association lists are frequently used for storing data that can be accessed by a *key*. This is similar to arrays or structures in other programming languages. The **subst** function, described later in this chapter, provides a convenient means of replacing the value associated with one key in an association list.

(atan *num1 [num2]*)

If *num2* is not supplied, **atan** returns the arctangent of *num1*, in radians. *num1* can be negative; the range of angles returned is –π/2 to +π/2 radians. For example:

```
(atan 0.5)                            returns   0.463648
(atan 1.0)                            returns   0.785398
(atan -1.0)                           returns   -0.785398
(angtos (atan -1.0) 0 4)              returns   "315.0000"
```

If both *num1* and *num2* are supplied, it returns the arctangent of *num1*/*num2*, in radians. If *num2* is zero, it returns an angle of plus or minus 1.570796 radians (+90° or –90 °), depending on the sign of *num1*. For example:

```
(atan 2.0 3.0)                   returns   0.588003
(angtos (atan 2.0 3.0) 0 4)      returns   "33.6901"
(atan 2.0 -3.0)                  returns   2.55359
(angtos (atan 2.0 -3.0) 0 4)     returns   "146.3099"
(atan -2.0 3.0)                  returns   -0.588003
(atan -2.0 -3.0)                 returns   -2.55359
(atan 1.0 0.0)                   returns   1.5708
(angtos (atan 1.0 0.0) 0 4)      returns   "90.0000"
(atan -0.5 0.0)                  returns   -1.5708
(angtos (atan -0.5 0.0) 0 2)     returns   "270.00"
```

Note: The **angtos** calls shown above illustrate a method of converting the radian value returned from **atan** into a string value.

(atof *string*)

This function returns the conversion of *string* into a real. For example:

```
(atof "97.1")          returns     97.1
(atof "3")             returns     3.0
```

(atoi *string*)

This function returns the conversion of *string* into an integer.

```
(atoi "97")            returns     97
(atoi "3")             returns     3
(atoi "3.9")           returns     3
```

(atom *item*)

This function returns `nil` if *item* is a list, and `T` otherwise. Anything that's not a list is considered an atom. For example, given the assignments

```
(setq a '(x y z))
(setq b 'a)
```

then

```
(atom 'a)              returns     T
(atom a)               returns     nil
(atom 'b)              returns     T
(atom b)               returns     T
(atom '(a b c))        returns     nil
```

Some versions of LISP differ in their interpretation of **atom**, so take care when using converted code.

(atoms-family *format [symlist]*)

This function returns a list of the built-in symbols and any other symbols defined in the current session. The first argument *format* is an integer value of 0 or 1. If the value of *format* is 0, a list of the currently defined symbols is returned; if *format* is 1, the symbols in the list are returned as strings.

```
(atoms-family 0)
```
returns a list of the currently defined symbols

The **atoms-family** function will also search for a list of symbol names. The *symlist* argument, if supplied, must be a list of strings that specify the symbol names to search for. If you want to verify that the symbols "CAR", "CDR", and "XYZ" have been defined, and you want the list returned as strings, you enter the following:

```
(atoms-family 1 '("CAR" "CDR" "XYZ"))
```

The **atoms-family** function returns a list of the type specified by format (symbols or strings) containing the names of the symbols that are defined and nil for those that are not defined. Assuming the symbol "XYZ" has not been defined, the following list is returned:

```
("CAR" "CDR" nil)
```

Caution: Prior to Release 12 of AutoCAD, atomlist was a symbol that could be redefined (chopped) or deleted. If you have any AutoLISP routines that used atomlist as a symbol, they will no longer function properly.

(Boole *func int1 int2 ...*)

This is a general bitwise Boolean function. The *func* argument is an integer between 0 and 15 representing one of the 16 possible Boolean functions in two variables. Successive integer arguments are bitwise (logically) combined based on this function and the following truth table:

Table 4–2. Boolean truth table

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

Each bit of *int1* pairs with the corresponding bit of *int2*, selecting one horizontal row of the truth table. The result bit is either 0 or 1, depending on the setting of the *func* bit corresponding to this row of the truth table.

If the appropriate bit is set in *func*, the result bit is 1, otherwise the result bit is 0. Some of the values for *func* are equivalent to the standard Boolean operations AND, OR, XOR, and NOT. These are shown here:

Table 4–3. Boole function bit values

Func	Operation	Result 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	NOT	Both input bits are zero (1's complement)

Examples

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

```
(Boole 1 12 5)          returns      4
```

Similarly, to specify a logical XOR of the values 6 and 5:

```
(Boole 6 6 5)           returns      3
```

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

```
(Boole 4 3 14)          returns      12
```

(boundp *atom*)

This function returns `T` if *atom* has a value bound to it (regardless of scope). If no value is bound to *atom* (or it has been bound to `nil`), **boundp** returns `nil`. If *atom* is an undefined symbol, it is automatically created and is bound to `nil`. For example, given these assignments:

```
(setq a 2) (setq b nil)
```

then

```
(boundp 'a)             returns      T
(boundp 'b)             returns      nil
```

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

(car *list*)

This function returns the first element of *list*. If *list* is empty, it returns `nil`. For example:

```
(car '(a b c))          returns      A
(car '((a b) c))        returns      (A B)
(car '())               returns      nil
```

(cdr *list*)

This function returns a list containing all but the first element of `list`. If `list` is empty, it returns `nil`. For example:

`(cdr '(a b c))`	returns	`(B C)`
`(cdr '((a b) c))`	returns	`(C)`
`(cdr '())`	returns	`nil`

When the `list` argument is a dotted pair (see the **cons** function on page 100), **cdr** returns the second element without enclosing it in a list. For example:

`(cdr '(a . b))`	returns	`B`
`(cdr '(1 . "Text"))`	returns	`"Text"`

(caar *list*), (cadr *list*), (cddr *list*), (cadar *list*), etc.

AutoLISP supports concatenations of **car** and **cdr**, up to four levels deep. For example, given this assignment:

```
(setq x '((a b) c d))
```

then

`(caar x)`	is equivalent to	`(car (car x))`	returning	`A`
`(cdar x)`	is equivalent to	`(cdr (car x))`	returning	`(B)`
`(cadar x)`	is equivalent to	`(car (cdr (car x)))`	returning	`B`
`(cadr x)`	is equivalent to	`(car (cdr x))`	returning	`C`
`(cddr x)`	is equivalent to	`(cdr (cdr x))`	returning	`(D)`
`(caddr x)`	is equivalent to	`(car (cdr (cdr x)))`	returning	`D`

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). Likewise, **caddr** can be used to obtain the *Z* coordinate of a 3D point. For instance, given these assignments:

`(setq pt2 '(5.25 1.0))`	*a 2D point*
`(setq pt3 '(5.25 1.0 3.0))`	*a 3D point*

then

`(car pt2)`	returns	`5.25`
`(cadr pt2)`	returns	`1.0`
`(caddr pt2)`	returns	`nil`
`(car pt3)`	returns	`5.25`
`(cadr pt3)`	returns	`1.0`
`(caddr pt3)`	returns	`3.0`

(chr *integer*)

This function returns the conversion of an integer representing an ASCII character code into a single-character string (similar to the **chr$** function in the BASIC language). For example:

`(chr 65)`	returns	`"A"`
`(chr 66)`	returns	`"B"`
`(chr 97)`	returns	`"a"`

(close *file-desc*)

This function closes a file and returns `nil`. The *`file-desc`* argument is a file descriptor obtained from the **`open`** function. After a **`close`**, the file descriptor is unchanged, but is no longer valid.

For example, assuming that `x` is a valid open file descriptor:

`(close x)` closes the associated file and returns `nil`

(command *[arguments]* ...)

This function executes AutoCAD commands from within AutoLISP and always returns `nil`. The arguments represent AutoCAD commands and their subcommands.

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.

```
(setq pt1 '(1 1) pt2 '(1 5))
(command "line" pt1 pt1 "")
```

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

Commands executed from the **`command`** function are not echoed to the screen if the AutoCAD system variable CMDECHO (accessible from **`setvar`** and **`getvar`**) is set to zero. The **`command`** function is the basic method of AutoCAD command access from AutoLISP.

Note: The **`get`*`xxx`*** user-input functions (**`getangle`**, **`getstring`**, **`getint`**, **`getpoint`**, etc.) cannot be used inside the **`command`** function. An attempt to do so results in the message:

```
error: AutoCAD rejected function
```

and termination of the function in progress. If user input is needed, issue the **`get`*`xxx`*** functions beforehand, or place them between successive **`command`** function calls.

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" on page 22.

Restrictions: The AutoCAD DTEXT and SKETCH commands read the keyboard and 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.

If an AutoCAD command is in progress and the predefined symbol PAUSE is encountered as an argument to the **command** function, the **command** function is suspended to allow direct user input.

- The PAUSE symbol is defined as a string consisting of a single backslash. You can use a backslash directly, rather than using the PAUSE symbol. However, if the **command** function is invoked from a menu item, the backslash suspends the reading of the menu item, which results in partial evaluation of the AutoLISP expression. Also, the pause mechanism might require a different trigger value in future versions of AutoLISP, so we recommend always using the PAUSE symbol rather than an explicit backslash.

  Reminder: When a backslash is used in a string, it must be preceded by another backslash (i.e., "\\").

- If PAUSE is encountered when a command is expecting input of a text string or an Attribute value, AutoCAD pauses for input only if the system variable TEXTEVAL is nonzero. Otherwise, the value of the PAUSE symbol (a single backslash) is taken as the text and does not cause a pause for input.
- When the **command** function has paused for user input, the function is still considered active, so the user cannot enter another AutoLISP expression to be evaluated.

Following is an example of a use of the PAUSE symbol:

```
(setq blk "MY_BLOCK")
(setq old_lay (getvar "clayer"))
(command "layer" "set" "NEW_LAY" "")
(command "insert" blk pause "" "" pause)
(command "layer" "set" old_lay "")
```

The preceeding code fragment would set the current layer to NEW_LAY, pause for user selection of an insertion point for the Block, MY_BLOCK—which would be inserted with *X* and *Y* scale factors of 1—and pause again for user selection of a rotation angle; the current layer would then be reset to the original layer.

If the **command** function specifies a PAUSE to the SELECT command and a PICKFIRST set is active, the SELECT command obtains the PICKFIRST set without pausing for the user.

Note: The Radius and Diameter subcommands of the Dim: prompt issue additional prompts in some situations. This can cause a failure of AutoLISP programs written prior to Release 11 that use these commands.

See also: "Command Submission" on page 21 for more information on the **command** function.

(cond (*test1 result1* ...) ...)

This 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 which follow the test that succeeded and returns the value of the last expression in the sublist. If there is only one expression in the sublist (i.e., *`result`* is missing), the value of the *`test`* expression is returned. The **cond** function is the primary conditional function in AutoLISP.

For example, the following 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 *new-first-element list*)

This is the basic list CONStructor. It takes an element (*`new-first-element`*) and a *`list`*, and returns the addition of that element to the beginning of the list. For example:

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

The first element can be an atom or a list.

The **cons** function also accepts an atom in place of the *`list`* argument, constructing a structure known as a *dotted pair.* When displaying a dotted pair, AutoLISP prints a period, or dot, between its first and second elements. You can use the **cdr** function to return the second atom of a dotted pair. Thus

```
(cons 'a 2)               returns        (A . 2)
(car (cons 'a 2))         returns        A
(cdr (cons 'a 2))         returns        2
```

A dotted pair is a special kind of list, and is not accepted as an argument by some functions that handle ordinary lists.

(cos *angle*)

This function returns the cosine of *`angle`*, where *`angle`* is expressed in radians. For example:

```
(cos 0.0)                 returns        1.0
(cos pi)                  returns        -1.0
```

(cvunit *value from to*)

This function converts a value or point from one unit of measurement to another. If successful, it returns the converted value or point. If either unit name is unknown (not found in the *acad.unt* file) or if the two units are dimensionally incompatible (as in converting grams into years), `nil` is returned.

The *`value`* argument is the numeric value you want to convert. It can also be a list containing two or three numbers to be converted (a 2D or 3D point). The *`from`* argument is the unit that the value is being converted from and *`to`* is the unit that the value is being converted into. The *`from`* and *`to`* arguments can name any unit type found in the *acad.unt* file.

Examples

```
(cvunit 1 "minute" "second")      returns     60.0
(cvunit 1 "gallon" "furlong")     returns     nil
(cvunit 1.0 "inch" "cm")          returns     2.54
(cvunit 1.0 "acre" "sq yard")     returns     4840.0
(cvunit '(1.0 2.5) "ft" "in")     returns     (12.0 30.0)
(cvunit '(1 2 3) "ft" "in")       returns     (12.0 24.0 36.0)
```

Suggestion: 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: "Real-world Units" on page 35.

(defun *sym argument-list expr ...*)

The **`defun`** function defines a function with the name *`sym`* (the function name is automatically quoted and must not be explicitly quoted). Following the function name is a list of arguments (possibly void), optionally followed by a slash and the names of one or more local symbols for the function. The slash must be separated from the first local symbol and from the last argument, if any, by at least one space. If you don't declare any arguments or local symbols, you must supply an empty set of parentheses after the function name.

Examples

The following *`argument-list`* examples show valid and invalid values:

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

You cannot define a function with multiple arguments of the same name, but you can have one that defines a local variable with the same name as another local variable or one of the arguments, as in:

```
(defun fubar (a a / b) ...)         Is not legal
(defun fubar (a b / a a b) ...)     Is fine
```

Following the list of arguments and local symbols are one or more expressions to be evaluated when the function is executed.

Note: If the argument/symbol list contains duplicate entries, the first occurrence of each name is used and the following occurrences are ignored.

The **defun** function returns the name of the function being defined. When the function so defined is invoked, its arguments are evaluated and bound to the argument symbols. The local symbols can be used within the function without changing their bindings at outer levels. The function returns the result of the last expression evaluated. All previous expressions in the function have only side effects. The **defun** function returns the name of the function defined.

The following examples define new functions with **defun** and show the values returned by the new functions:

```
(defun add10 (x)
   (+ 10 x)
)                            returns      ADD10
(add10 5)                    returns      15
(add10 -7.4)                 returns      2.6
```

and

```
(defun dots (x y / temp)
   (setq temp (strcat x "..."))
   (strcat temp y)
)                            returns      DOTS
(dots "a" "b")               returns      "a...b"
(dots "from" "to")           returns      "from...to"
```

Warning: Never use the name of a built-in function or symbol as *sym*, since this makes the built-in function inaccessible. To get a list of built-in and previously defined functions, see page 95 for information on the **atoms-family** function.

Related topics: See "Defining Functions and Automatic Loading" on page 14 for information on function libraries. Also, see the **setq** function for examples of local and global symbols.

(distance *pt1 pt2*)

This function returns the 3D distance between the points *pt1* and *pt2*. The following examples demonstrate this:

```
(distance '(1.0 2.5 3.0) '(7.7 2.5 3.0))      returns   6.7
(distance '(1.0 2.0 0.5) '(3.0 4.0 0.5))      returns   2.82843
```

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.

See also: "Geometric Utilities" on page 25.

(distof *string [mode]*)

Converts *string*, which contains a real (floating-point) value in the display format specified by *mode*, into a real value.

The *mode* argument specifies the units in which the string is formatted. The value should correspond to values allowed for the AutoCAD system variable LUNITS, as shown in the following table. Also, if *mode* is omitted, **distof** uses the current value of LUNITS.

Table 4–4. Linear units values

Mode value	String format
1	Scientific
2	Decimal
3	Engineering (feet and decimal inches)
4	Architectural (feet and fractional inches)
5	Fractional

The argument *string* must be a string that **distof** can parse correctly according to the mode specified by *mode*. It can be in the same form that **rtos** would return, or in a form that AutoCAD allows for keyboard entry. 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).

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.

If **distof** succeeds, it returns a real number; otherwise, it returns `nil`.

(entdel *ename*)

The entity specified by *ename* is deleted if currently in the drawing, and *undeleted* (restored to the drawing) if it has been deleted previously in this editing session. Deleted entities are purged from the drawing when leaving the drawing editor, so **entdel** can only restore them during the editing session in which they were deleted.

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 achieve this).

You cannot delete entities within a Block Definition. However, you can completely redefine a Block Definition (minus the entity you want deleted) using **entmake** to accomplish this.

Example

```
(setq e1 (entnext))     Sets e1 to the name of the first entity in the drawing
(entdel e1)             Deletes entity e1
(entdel e1)             Undeletes (restores) deleted entity e1
```

(entget *ename [applist]*)

The entity whose name is `ename` is retrieved from the database and returned as a list containing its definition data. If an optional list of registered application names (`applist`) is supplied, the extended entity data associated with the specified applications is also returned. The data is coded as a LISP association list, from which you can extract items by using the **assoc** function. Objects in the list are assigned AutoCAD DXF group codes for each part of the entity data.

These assumptions apply in the following example:

- the current layer is 0
- the current linetype is CONTINUOUS (the default)
- the current elevation is zero (the default), and
- entity handles are disabled

If you draw a Line with the following sequence of commands:

```
Command: line
From point: 1,2
To point: 6,6
To point: ↵
```

then you can retrieve the entity data for the Line by entering this:

Command: **(setq a (entget (entlast)))***which sets* a *equal to the list:*

```
(   (-1 . <Entity name: 60000014>)
    (0 . "LINE")                                Entity type
    (8 . "0")                                   Layer
    (10 1.0 2.0 0.0)                            Start point
    (11 6.0 6.0 0.0)                            Endpoint
)
```

The –1 item at the start of the list contains the name of the entity this list represents. The **entmod** function described later uses it to identify the entity to be modified.

The individual dotted pairs that represent the values can be easily extracted by **assoc**, with **cdr** used to pull out their values. The codes for the components of the entity are those used by DXF and documented in chapter 11 of the *AutoCAD Customization Manual.*

As with DXF, entity header items (color, linetype, thickness, the attributes-follow flag, and the entity handle) are output only if they have nondefault values. *Unlike* DXF, optional entity definition fields are output whether equal to their defaults or not. This simplifies processing; programs can always assume these fields to be present for general algorithms that operate on them. *Also* unlike DXF, associated *X*, *Y*, and *Z* coordinates are grouped together into one point list, as in (10 1.0 2.0 3.0), rather than appearing as separate 10, 20, and 30 groups.

The sublists for points are not dotted pairs like the rest. The convention is that the **cdr** of the sublist is the group's value. Since a point is a list of two (or three) reals, that makes the entire group a three (or four) element list. The **cdr** of the group is the list representing the point, so the convention that **cdr** always returns the value is preserved.

When writing functions to process these entity lists, be sure to make them insensitive to the order of the sublists. Use **assoc** to guarantee this. The –1 group containing the entity's name allows modification operations to accept the entity list and avoids the need to keep the entity name in a parallel structure. A Seqend entity at the end of a Polyline or a set of Attributes contains a –2 group whose **cdr** is the entity name of the header of this entity. This allows the header to be found from a subentity by walking forward to the Seqend, and then using the **cdr** of the –2 group as the entity name to retrieve the associated main entity.

The following example illustrates a more complex entity's representation as a list. For this example, we'll assume the current UCS is rotated 40 degrees counterclockwise about the *X* axis of the WCS and that entity handles are enabled.

```
Command: linetype
?/Create/Load/Set: set
New entity linetype <BYLAYER>: dashed
?/Create/Load/Set: ↵
Command: color
New entity color <BYLAYER>: blue
Command: layer
?/Make/Set/New/On/Off/Color/Ltype/Freeze/Thaw: make
New current layer <0>: annotation
?/Make/Set/New/On/Off/Color/Ltype/Freeze/Thaw: ↵
Command: text
Start point or Align/Center/Fit/Middle/Right/Style: 2,2
Height <0.2000>: .3
Rotation angle <0>: 30
Text: So long, and thanks for all the fish!
Command: (setq ed (entget (setq e (entlast))))
```

In this case, e is set to the Text entity's name, and ed is set to the list that follows. Examining chapter 11 of the *AutoCAD Customization Manual* should clarify the meaning of this list.

```
(   (-1 . <Entity name: 6000053C>)
    (0 . "TEXT")                                   Entity type
    (8 . "ANNOTATION")                             Layer
    (6 . "DASHED")                                 Linetype
    (62 . 5)                                       Color
    (5 . "7E")                                     Handle
    (10 2.0 2.0 0.0)                               Start point
    (40 . 0.3)                                     Height
    (1 . "So long, and thanks for all the fish!")
    (50 . 0.523599)                                Rotation angle (radians)
    (41 . 1.0)                                     Width factor
    (51 . 0.0)                                     Obliquing angle
    (7 . "STANDARD")                               Text style
    (71 . 0)                                       Generation flags
    (72 . 0)                                       Horizontal justification
    (73 . 0)                                       Vertical justification
    (11 0.0 0.0 0.0)                               Alignment point
    (210 0.0 -0.642788 0.766044)                   Extrusion direction vector
)
```

All points associated with an entity are expressed in terms of that entity's Entity Coordinate System (ECS). For Point, Line, 3D Line, 3D Face, 3D Polyline, 3D Mesh, and Dimension entities, the ECS is equivalent to the WCS (the entity points are World points). For all other entities, the ECS can be derived from the WCS and the entity's extrusion direction (its 210 group). When working with entities that have been drawn using coordinate systems other than the WCS (such as the Text in the above example), you might need to convert the points to the WCS or to the current UCS by using the **trans** function. Using the above Text entity as an example:

```
(setq p (cdr (assoc 10 ed)))   returns   (2.0 2.0 0.0)
```

setting `p` to the text start point, in terms of the Text entity's ECS. (Note that the point is returned in this manner regardless of the current UCS setting at the time of the **entget**.) Now:

```
(trans p e 0)       returns     (2.0 1.53209 1.28558)
```

This uses `e` (the Text entity name) as the *from* conversion code, and translates the text start point from the Text's ECS to World coordinates.

Before performing an `(entget)` or `(entmod)` on Vertex entities, you should read or write the header (Polyline entity) for the Polyline to which they belong. If the Polyline entity most recently processed is different from the one to which the Vertex belongs, width information (the 40 and 41 groups) can be lost.

(entlast)

This function returns the name of the last nondeleted main entity in the database. This function is frequently used to obtain the name of a new entity which has just been added via the **command** function. The entity need not be on screen nor on a thawed layer to be selected.

For example:

```
(setq e1 (entlast))         Sets e1 to the name of the last main entity in the drawing
(setq e2 (entnext e1))      Sets e2 to nil (or to an Attribute or Vertex subentity name)
```

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,
         (setq a b)                             loops until no more
      )
   )
   a                                            Returns last main/subentity
)
```

(entmake *[elist]*)

This function creates a new entity in the drawing. If the entity is successfully created, its list of definition data is returned. If the entity cannot be created for some reason (like incorrect supplied data), it returns `nil`.

The *`elist`* argument must be a list of the entity's definition data in a format similar to that returned by the **`entget`** function. The *`elist`* 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.

One method of creating a new entity is by obtaining an entity's definition data with the **`entget`** function, modifying it, and then appending a new entity to the drawing with the **`entmake`** function.

Before creating a new entity, **`entmake`** verifies that a valid layer name, linetype name, and colour are supplied. If a new layer name is introduced, **`entmake`** automatically creates the new layer. The **`entmake`** function also checks for Block names, Dimstyle names, Text style names, and Shape names if the entity type requires them.

The entity type (e.g., Circle, Line) must be the first or second field of the *`elist`*. If it is the second field, it can only be preceded by the entity name. This is the format returned by **`entget`**. In such cases, it ignores the entity name when the new entity is created. If the *`elist`* contains an entity handle, it also is ignored.

This example creates a red circle on your drawing with its centre point at coordinate (4,4) and a radius of 1. The optional layer and linetype fields have been omitted and therefore assume default values.

```
(entmake    '( (0 . "CIRCLE")              Entity type
               (62 . 1)                     Color
               (10 4.0 4.0 0.0)             Center point
               (40 . 1.0)                   Radius
             )
)
```

Note: Entities created on a frozen layer are not regenerated until the layer is thawed.

Complex Entities

A complex entity (a Block Definition, a Polyline, or a Block Reference containing Attributes) can be created by several **`entmake`** calls to define its subentities (Attributes or Vertices). When **`entmake`** sees that a complex entity is being created, it creates a temporary file to gather the definition data. For each **`entmake`**, a check is performed to see if the temporary file exists (meaning a complex entity is being defined); if so, the new data is appended to the file. When the definition of the complex entity is complete (by appending the appropriate Seqend or Endblk entity), the supplied data is rechecked and the complex entity is added to the drawing. Completion of a Block definition (**`entmake`** of an Endblk) returns the Block's name rather than the entity data list normally returned.

If data is received during the creation of a complex entity that is invalid for that entity type, the entity is rejected as well as the entire complex entity. A

Block definition cannot be nested, nor can it reference itself. However, a Block definition can contain references to other Block definitions.

A group 66 code is only honoured for Insert entities (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 zero. The only entity that can follow a Polyline entity is a Vertex entity.

No portion of a complex entity is displayed on your drawing until its definition is complete. You can cancel the creation of a complex entity by entering **entmake** with no arguments. This clears the temporary file and returns `nil`.

All entities of a complex entity must have the same space setting. The entities can exist in either model space or in paper space, but not both. For example, Polyline, Vertex, and Seqend entities must be in the same space. The same is true for Insert, Attrib, and Seqend entities.

The Block and Endblk entities can be used to create a new Block definition. Newly created Blocks are automatically entered into the symbol table where they can be referenced.

Applications might want to represent polygons with an arbitrarily large number of sides in Polyface meshes. However, the AutoCAD entity structure imposes a limit on the number of vertices that a given face entity can specify. You can represent more complex polygons by decomposing them into triangular wedges. AutoCAD represents triangular wedges as 4-vertex Faces where two adjacent vertices have the same value. Their edges should be made invisible to prevent visible artifacts of this subdivision from being drawn. The PFACE command performs this subdivision automatically, but when applications generate polyface meshes directly, the applications must do this themselves.

The number of vertices per face is the key parameter in this subdivision process. The PFACEVMAX system variable provides an application with the number of vertices per face entity. This value is read-only and is set to 4.

Important: You cannot **entmake** Viewport entities.

Caution: When **entmake** creates a Block, it can overwrite an existing Block. The **entmake** function does not check for name conflicts in the Block Definitions table, so before you use it to create a named Block, you should use **tblsearch** (described on page 161) to ensure that the name of the new Block is unique. However, using **entmake** to redefine anonymous blocks (described in the next section) can be useful.

Anonymous Blocks

The Block Definitions table in a drawing can contain anonymous Blocks. Anonymous Blocks are created to support hatch patterns and associative dimensioning. They can also be created by **entmake** for the application's own purposes, usually to contain entities that the user cannot access directly.

The name (group 2) of an anonymous Block is ∗U*nnn*, where *nnn* is a number generated by AutoCAD. Also, the low-order bit of an anonymous block's *Block type flag* (group 70) is set to one. When **entmake** creates a block whose name begins with * and whose anonymous bit is set, AutoCAD treats this as an anonymous block and assigns it a name. Characters following the * in the name string passed to **entmake** are *ignored*. After the Block is created, **entmake** returns its name. If you are creating the block by multiple **entmake** calls, it returns the name after a successful call of:

```
(entmake "endblk")
```

Whenever a drawing is brought into the drawing editor, all unreferenced anonymous Blocks are purged from the Block Definitions table. Referenced (Inserted) anonymous blocks are not purged. You can use **entmake** to create a block reference (Insert) to an anonymous block (you cannot pass an anonymous block to the Insert command). You can also use **entmake** to redefine the block. The entities in a block (but not the Block entity itself) can be modified with **entmod**.

Caution: Although a referenced anonymous block becomes permanent, the numeric portion of its name can change between drawing editor sessions. Applications cannot rely on anonymous block names remaining constant.

(entmod *elist*)

The **entmod** function is passed a list (`elist`) in the format returned by **entget**, and updates the database information for the entity whose name is specified by the –1 group in `elist`. Therefore the primary mechanism through which AutoLISP updates the database is by retrieving entities with **entget**, modifying the list defining an entity (note that the AutoLISP **subst** function is extremely useful for this), and updating the entity in the database with **entmod**.

Example

```
(setq en (entnext))        Sets en to the name of the first entity in the drawing
(setq ed (entget en))      Sets ed to the entity data for entity name en
(setq ed
   (subst(cons 8 "0")
         (assoc 8 ed)      Changes the layer group in ed
         ed                to layer 0
   )
)
(entmod ed)                Modifies entity en's layer in drawing
```

The **entmod** function imposes some restrictions on the changes it makes. First of all, an entity's type and handle cannot be changed. (If you want to do this, just **entdel** it and make a new entity with the **command** or **entmake** functions.) All objects referenced by the entity list must be known to AutoCAD before the **entmod** is executed. Thus Text style, Linetype, Shape, and Block names must have been previously defined in a drawing before they can be used in an entity list with **entmod**. An exception to this is layer names—**entmod** creates a new layer with the standard defaults used by the LAYER New command if a previously undefined layer is named in an entity list.

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 colour number), **entmod** truncates it and converts it to an integer.

The **entmod** function performs the same consistency checking on the list supplied to it as DXFIN does on the data from a DXF file. If a serious error is detected, the database is not updated and `nil` is returned. Otherwise, **entmod** returns the list given to it as its argument. **entmod** cannot change internal fields such as the entity name in the –2 group of a Seqend entity—attempts to change such fields are ignored.

When **entmod** updates a main entity, it modifies the entity and updates its image on screen (including subentities). When **entmod** updates a subentity (a Polyline vertex or a Block attribute), the subentity is updated in the database but the image on screen is not redisplayed. After all modifications are made to a given entity's subentities, the **entupd** function described later can be used to update the image on screen.

Important: 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 entities). If you **entmod** an entity within a Block Definition, the modification will affect all instances of the Block in the drawing.

Before performing an `(entget)` or `(entmod)` on Vertex entities, you should read or write the header (Polyline entity) for the Polyline to which they belong. If the Polyline entity most recently processed 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, thereby affecting all insertions of the block. By doing so it is possible to create a self-referencing Block. Doing this can cause AutoCAD to crash.

(entnext *[ename]*)

If called with no arguments, this function returns the entity name of the first nondeleted entity in the database. If **entnext** is called with an entity name argument *ename*, it 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.

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 have obtained the name of a subentity via **entnext**, you can find the parent entity by walking forward via **entnext** until a Seqend entity is found, then extracting the –2 group from that entity, which is the main entity's name. Examples follow:

`(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`

(entsel *[prompt]*)

Sometimes when operating on entities, you want to simultaneously select an entity 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 entity, requiring the selection to be by a point pick. The current Osnap setting is ignored by this function (no object snap) unless you specifically request it while you are in the function. The **entsel** function honours keywords from a preceding call to **initget**.

The `entsel` function returns a list whose first element is the entity name of the chosen entity, and whose second element is the coordinates (in terms of the current UCS) of the point used to pick the entity. If a string is specified for *`prompt`*, that string is used to ask the user for the entity. Otherwise, the prompt defaults to Select object:. 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: ↵
Command: (setq e (entsel "Please choose an entity: "))
Please choose an entity: 3,3
(<Entity name: 60000014> (3.0 3.0 0.0))
```

A list of the form returned by `entsel` can be supplied to AutoCAD in response to any of its object selection prompts. It is treated by AutoCAD as a pick of the designated entity by pointing to the specified point.

Related topics: See `initget` on page 128.

(entupd *ename*)

When a Polyline vertex or Block attribute is modified with `entmod`, the entire complex entity is not updated on screen. If, for example, 100 vertices of a complex Polyline were to be modified, recalculating and redisplaying the Polyline as each vertex was changed would be unacceptably slow. The `entupd` function can be used to cause a modified Polyline or Block to be updated on 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—`entupd` will find the head. 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, all entities might not be regenerated. To ensure complete regeneration, you must invoke the REGEN command. You can do this from AutoLISP by entering `(command "regen")`.

Example

Assuming that the first entity in the drawing is a Polyline with several vertices, then

```
(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
```

(eq *expr1 expr2*)

This function determines whether `expr1` and `expr2` are identical; that is, whether they are actually bound to the same object (by **setq**, for example). **eq** returns `T` if the two expressions are identical, and `nil` otherwise. It is typically used to determine whether two lists are actually the same. For example, given the following assignments:

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

then

`(eq f1 f3)`	returns	`nil`	`f1` *and* `f3` *are not the same list!*
`(eq f3 f2)`	returns	`T`	`f3` *and* `f2` *are exactly the same list*

Related topics: Compare this function with the = function on page 88 and the **equal** function described next.

(equal *expr1 expr2 [fuzz]*)

This function determines whether `expr1` and `expr2` are equal; that is, whether they evaluate to the same thing. For example, given the following assignments:

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

then

`(equal f1 f3)`	returns	`T`	`f1` *and* `f3` *evaluate to the same thing*
`(equal f3 f2)`	returns	`T`	`f3` *and* `f2` *are exactly the same list*

Although two lists that the **equal** function finds the same might not be found so using the **eq** function, atoms that are found to be the same using the **equal** function are always found to be the same if you use the **eq** function. If the **eq** function finds the list or atoms the same, the **equal** function also always finds them the same.

When comparing two real numbers (or two lists of real numbers, as in points), you should realize that two *identical* numbers can differ slightly if different methods are used to calculate them. Therefore, an optional numeric argument, `fuzz`, lets you specify the maximum amount by which `expr1` and `expr2` can differ and still be considered **equal**.

For example, given

```
(setq a 1.123456)
(setq b 1.123457)
```

then

`(equal a b)`	returns	`nil`
`(equal a b 0.000001)`	returns	`T`

Related topics: Compare this function with the = function on page 88 and the **eq** function described previously.

(*error* *string*)

This is a user-definable error handling function. If it is not `nil`, it is executed as a function whenever an AutoLISP error condition exists. It is passed one argument, a string containing a description of the error.

Example

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

This function would do exactly the same thing that the AutoLISP standard error handler would do; print error: and the description.

(eval *expr*)

Returns the result of evaluating *expr*, where `expr` is any AutoLISP expression. For example, given these assignments:

```
(setq a 123)
(setq b 'a)
```

then

```
(eval 4.0)            returns     4.0
(eval (abs -10))      returns     10
(eval a)              returns     123
(eval b)              returns     123
```

(exit)

The **`exit`** function forces the current application to quit. If **`exit`** is called, it returns the error message quit/exit abort and returns to the AutoCAD Command: prompt.

See also: The **`quit`** function on page 146.

(exp *number*)

This function returns the constant *e* raised to the power of *`number`* (the natural antilog). It returns a real. For example:

```
(exp 1.0)             returns     2.71828
(exp 2.2)             returns     9.02501
(exp -0.4)            returns     0.67032
```

(expand *number*)

Allocates node space by requesting a specified *`number`* of segments. See "Manual Allocation" on page 179 for more information on **`expand`**.

(expt *base power*)

This function returns `base` raised to the specified `power`. If both arguments are integers, the result is an integer. Otherwise the result is a real. These are examples:

```
(expt 2 4)          returns     16
(expt 3.0 2.0)      returns     9.0
```

(findfile *filename*)

The **findfile** function searches the AutoCAD library path for the file specified by `filename` and, if found, returns a fully qualified path/filename.

The AutoCAD library path is searched in the following order:

1. The current directory
2. The directory containing the current drawing file
3. The directories named by the ACAD environment variable (if this variable has been specified)
4. The directory containing the AutoCAD program files

Note: Depending on the current environment, two or more of these directories might be the same.

The **findfile** function makes no assumption about the file type or extension of `filename`; if you want one, you must supply it. If the name is not qualified (i.e., if it does not have a drive/directory prefix), AutoCAD searches for it and returns the fully qualified name, or `nil` if the file is not found. If a drive/directory prefix is supplied, AutoCAD looks only in that directory (performing no library search). The following examples use / as the directory separator; on DOS systems, you can use either / or \.

For example, if the current directory is */acad* and contains the file *abc.lsp,* we are editing a drawing in the */acad/drawings* directory, the ACAD environment variable is set to */acad/support,* the file *xyz.txt* exists only in the */acad/support* directory, and the file *nosuch* is not present in any of the directories on the library search path; then

```
(findfile "abc.lsp")        returns     "/acad/abc.lsp"
(findfile "xyz.txt")        returns     "/acad/support/xyz.txt"
(findfile "nosuch")         returns     nil
```

The fully qualified name returned by **findfile** is suitable for use with the **open** function.

See also: "File Search" on page 24.

(fix *number*)

This function returns the conversion of *number* into an integer. The *number* argument can be either an integer or a real. If real, it is truncated to the nearest integer by discarding the fractional portion. For example:

```
(fix 3)                 returns      3
(fix 3.7)               returns      3
```

Note: 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).

(float *number*)

This function returns the conversion of *number* into a real. The *number* argument can be either an integer or a real. For example:

```
(float 3)               returns      3.0
(float 3.75)            returns      3.75
```

(foreach *name list expr ...*)

This function steps through *list* assigning each element to *name*, and evaluates each *expr* for every element in the list. Any number of *exprs* can be specified. The **foreach** function returns the result of the last *expr* evaluated. For example:

```
(foreach n '(a b c) (print n))
```

is equivalent to:

```
(print a)
(print b)
(print c)            and returns       c
```

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

(gc)

Forces a garbage collection, which frees up unused nodes. See "Node Space" on page 177 for greater explanation of garbage collection.

(gcd *num1 num2*)

This function returns the greatest common denominator of *num1* and *num2*. The *num1* and *num2* arguments must be integers. For example:

```
(gcd 81 57)             returns      3
(gcd 12 20)             returns      4
```

(getangle *[pt] [prompt]*)

This function pauses for user input of an angle and then returns that angle in radians. The **getangle** function measures angles with the zero-radian direction being the current angle set by the ANGBASE 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).

The *prompt* argument is an optional string to be displayed as a prompt, and *pt* is an optional 2D base point in the current UCS. The user 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 whatever, 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 crosshair position to help you visualize the angle. The **getangle** function's optional *pt* argument, if specified, is assumed to be the first of these two points, allowing the user to show AutoLISP the angle by pointing to one other point. You can supply a 3D base point, but this can be confusing since the angle is always measured in the current construction plane.

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

The following are example **getangle** calls:

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

The user cannot enter another AutoLISP expression as the response to a **getangle** request. An attempt to do so results in this message:

Can't reenter AutoLISP.

Related topics: See the illustration and comparison to **getorient** on page 121. Also see **initget** on page 128.

(getcorner *pt [prompt]*)

The **getcorner** function returns a point in the current UCS, similar to **getpoint**. However, **getcorner** requires a base point argument *pt* and draws a rectangle from that point as the user moves the crosshairs on screen. The *prompt* argument is an optional string to be displayed as a prompt.

The base point is expressed in terms of the current UCS. If the user supplies a 3D base point, its *Z* coordinate is ignored; the current elevation is used as the *Z* coordinate.

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

Related topics: See **getpoint** on page 122 and **initget** on page 128.

(getdist *[pt] [prompt]*)

The `getdist` function pauses for user input of a distance or one or two points and returns a real number that is the distance between those points.

The user can 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), this function always returns the distance as a real.

If the user picks two points, **getdist** returns the distance between them, drawing a rubber-band line from the first point to the current crosshair position to assist in visualizing the distance. The *pt* argument is an optional 2D or 3D base point in the current UCS. If provided, *pt* is used as the first of the two points and the user is prompted for only the second point.

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 return a 2D distance.

The *prompt* argument is an optional string to be displayed as a prompt.

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

The following are examples of how you can use the **getdist** function.

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

Related topics: See **initget** on page 128.

(getenv *variable-name*)

This function returns the string value assigned to a system environment variable. The *variable-name* argument is a string specifying the name of the variable to be read. If this variable does not exist, **getenv** returns `nil`.

For example, if the system environment variable ACAD is set to */acad/support* and there is no variable named NOSUCH, then:

```
(getenv "ACAD")      returns    "/acad/support"
(getenv "NOSUCH")    returns    nil
```

Note: On UNIX systems, ACAD and `acad` refer to two different environment variables, since these operating systems are case-sensitive.

(getfiled *title default ext flags*)

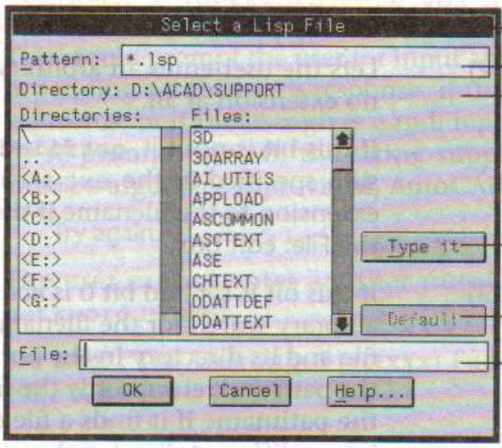
The **getfiled** function displays a dialogue box containing a list of available files of a specified extension type. You can use this to browse through different drives and directories, select an existing file, or specify the name of a new file.

This function prompts the user for a filename via the standard AutoCAD file dialogue box. The *title* argument specifies the label of the entire dialogue box, *default* specifies a default filename to use (which can be a null string [""]), and *ext* is the default filename extension (if passed as a null string [""], *ext* defaults to *). The following figure shows how these arguments affect the dialogue box's appearance. If the dialogue box obtains a filename from the user, **getfiled** returns a string that specifies the filename; otherwise, **getfiled** returns `nil`.

Example

The dialogue box shown next appears when the following call to **`getfiled`** is issued:

```
(getfiled "Select a Lisp File" "/acad/support/" "lsp" 8)
```



Select a Lisp File

Pattern: *.lsp

Directory: D:\ACAD\SUPPORT

Directories: \ .. <A:> <B:> <C:> <D:> <E:> <F:> <G:>

Files: 3D 3DARRAY AI_UTILS APPLOAD ASCOMMON ASCTEXT ASE CHTEXT DDATTDEF DDATTEXT

File:

set by the `title` argument

set by the `ext` argument

set by the pathname portion of the `default` argument— if `default` doesn't specify a path, this is initially the current directory

disabled if bit value 2 is used in `flags` argument

enabled if a default filename is supplied

set by the filename portion of the `default` argument

Figure 4–1. Sample getfile dialogue box

The *`flags`* argument is an integer value (a bit-coded field) that controls the behaviour of the dialogue box. To set more than one condition at a time, simply add the values together (in any combination) to create a *`flags`* value between 0 and 15. The *`flags`* argument values and meanings are as follows:

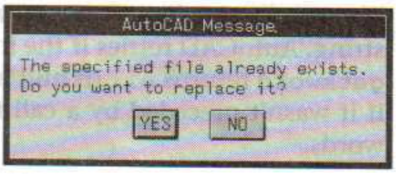
Table 4–5. getfiled flags options

Value	Meaning
1	Indicates a request for a new file to be created
2	Disables the Type it button
4	Lets the user enter an arbitrary filename extension
8	Performs a library search for the filename entered

The following describes the *`flags`* values in more detail:

Value =1 (bit 0) This bit should be set when you are prompting for the name of a new file to *create*. You should not set this bit when you are prompting 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 dialogue 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 cancelling the operation. The following figure shows the alert box and message.



AutoCAD Message

The specified file already exists.
Do you want to replace it?

Value = 2 (bit 1) Disables the Type it button. This bit is automatically set if **getfiled** is called while another dialogue box is active (otherwise, it would force the other dialogue box to disappear as well).

If this bit is not set, the Type it button is enabled. If the user selects it, the dialogue box disappears and **getfiled** returns a value of 1.

Value = 4 (bit 2) Lets the user enter an arbitrary filename 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 filename if the user doesn't enter it in the File: edit box.

Value = 8 (bit 3) If this bit is set and bit 0 is *not* set, **getfiled** performs a library search for the filename entered. If it finds the file and its directory in the library search path, it strips the path and returns only the filename. (It doesn't strip the pathname if it finds a file of the same name but it is in a different directory.)

If this bit is not set, **getfiled** returns the entire filename, including the pathname.

You should set this bit if you're using the dialogue box to open an existing file whose name you want to save in the drawing (or other database), and which you will search for later by calling **findfile**.

(getint *[prompt]*)

This function pauses for user input of an integer, and returns that integer. Values can range from –32,768 to +32,767. The *prompt* argument is an optional string to be displayed as a prompt. For example:

```
(setq num (getint))
(setq num (getint "Enter a number: "))
```

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

See also: "The User-input (getxxx) Functions" on page 29 and **initget** on page 128.

(getkword *[prompt]*)

The **getkword** function requests a keyword from the user. The list of valid keywords is set prior to the **getkword** call, using the **initget** function. The *prompt* argument is an optional string to be displayed as a prompt.

The **getkword** function returns the keyword matching the user input as a string. AutoCAD retries if the input is not a keyword. If the input is null (↵), **getkword** returns `nil` (if null input is allowed). This function also returns `nil` if it wasn't preceded by a call to **initget** that established one or more keywords.

Example

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

```
(initget 1 "Yes No")
(setq x (getkword "Are you sure? (Yes or No) "))
```

This would prompt the user for input and set the symbol x to either Yes or No, depending on the user's response. If the response does not match any of the keywords, or if the user gives a null reply, AutoCAD asks the user to try again by re-prompting with the string supplied in the *prompt* argument. If no *prompt* argument is provided, AutoCAD supplies this prompt:

```
Try again:
```

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

See also: "The User-input (getxxx) Functions" on page 29 and **initget** on page 128.

(getorient *[pt] [prompt]*)

This function is similar to the **getangle** function, except that the angle value returned by **getorient** is unaffected by the AutoCAD system variables ANGBASE and ANGDIR. The **getorient** function always measures angles with the zero-radian direction being to the right (east) and angles increasing in the counterclockwise direction. As with **getangle**, **getorient** expresses the returned angle in radians, with respect to the current construction plane.

The *pt* and *prompt* arguments are the same as in **getangle**.

You should understand the difference between the input angle and the angle returned by **getorient**. Angles input to **getorient** are based on the current settings of ANGDIR and ANGBASE. However, once an angle is input, 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 have selected a different zero-degree base or a different direction for increasing angles by using the UNITS command or the ANGBASE and ANGDIR system variables.

You should use **getangle** when you need a rotation amount (a relative angle), whereas you should use **getorient** to obtain an orientation (an absolute angle).

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

See also: "The User-input (getxxx) Functions" on page 29, **getangle** on page 116, and **initget** on page 128.

(getpoint *[pt] [prompt]*)

This function pauses for user input of a point. The *pt* argument is an optional 2D or 3D base point in the current UCS, and *prompt* is an optional string to be displayed as a prompt. The user can specify a point by pointing or by entering a coordinate in the current units format. If the optional *pt* base point argument is present, AutoCAD draws a rubber-band line from that point to the current crosshair position. For example:

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

The returned value is a 3D point expressed in terms of the current UCS.

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

See also: "The User-input (getxxx) Functions" on page 29, **getcorner** on page 116, and **initget** on page 128.

(getreal *[prompt]*)

This function pauses for user input of a real number and returns that real number. The *prompt* argument is an optional string to be displayed as a prompt. For example:

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

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

See also: "The User-input (getxxx) Functions" on page 29 and **initget** on page 128.

(getstring *[cr] [prompt]*)

The **getstring** function pauses for user input of a string, and returns that string. If the string is longer than 132 characters, it returns only the first 132 characters of the string. If the input string contains the backslash character (\), it is converted to two backslash characters (\\). This is done so the returned value can contain filename paths that can be used by other functions.

If *cr* is supplied and is not nil, the input string can contain blanks (and must therefore be terminated by a ↵). Otherwise, the input string is terminated by space or ↵. The *prompt* argument is an optional string to be displayed as a prompt.

Examples

```
(setq s (getstring "What's your first name? "))
```

responding **John** returns "John"

```
(setq s (getstring T "What's your full name? "))
```

responding **John Doe** returns "John Doe"

```
(setq s (getstring "Enter filename: "))
```

responding **\files\acad\mydwg** returns "\\files\\acad\\mydwg"

Note: If your routine expects the user to enter one of several known options (keywords), it can use the **getkword** function instead.

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

Related topics: See the description of the **getkword** function on page 120.

(getvar *varname*)

This function retrieves the value of an AutoCAD system variable. The variable name must be enclosed in double quotes. For example, assuming that the fillet radius specified most recently was 0.25 units:

```
(getvar "FILLETRAD")    returns    0.25
```

If you use **getvar** to retrieve the value of a system variable unknown to AutoCAD, it returns nil. You can find a list of the current AutoCAD system variables in appendix A of the *AutoCAD Reference Manual*.

Related topics: See **setvar** on page 152.

See also: "System and Environment Variables" on page 23.

(graphscr)

On single-screen AutoCAD installations, the **graphscr** function causes the display to switch from the text screen to the graphics screen. This is equivalent to the AutoCAD command GRAPHSCR or to pressing the Flip Screen function key (when the text screen is current).

The **textscr** function is the complement of **graphscr**.

The **graphscr** function always returns nil.

Related topics: See **textscr** and **textpage** starting on page 162.

(grclear)

This function clears the current viewport. (On single-screen systems, it flips to the graphics screen from the text screen first.) It leaves the command/prompt, status, and menu areas unchanged. You can use the **redraw** function to restore the prior contents of the graphics screen. The **grclear** function always returns nil.

(grdraw *from to color [highlight]*)

The **grdraw** function draws a vector between two points, in the current viewport. The `from` and `to` arguments are 2D or 3D points (lists of two or three reals) that specify the endpoints of the vector in terms of the current UCS. AutoCAD clips the vector as required to fit the screen. It draws the vector with the colour specified by the integer `color` argument, with –1 signifying *XOR ink*, which complements anything it draws over and erases itself when overdrawn.

If the optional integer `highlight` argument is supplied and is nonzero, the vector is drawn using the default highlighting method of the display device (usually dashed). If `highlight` is omitted or is supplied and is zero, **grdraw** uses the normal display mode.

You can use the **grvecs** function to draw multiple vectors on the graphics screen (see page 127).

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

The **grread** function directly reads the next input provided by the user to *any* of the AutoCAD input devices; it can optionally track the pointing device as it is moved. This is how AutoCAD implements dragging.

Caution: Only very specialized AutoLISP routines need this function: most input to AutoLISP should be obtained through the various **get*xxx*** functions such as **getstring**, **getreal**, and so on.

If the `track` argument is supplied and is not `nil`, it enables the return of coordinates from a pointing device as it is moved, and a selection button does not have to be pressed. The `allkeys` argument is an optional integer value (a bit-coded field). If the `allkeys` argument is present, **grread** will perform various functions depending on the code supplied. The `curtype` argument can be used to control the type of cursor displayed.

track Track cursor location if not `nil`.

allkeys If supplied, the `allkeys` argument must be an integer and is defined as follows:

Value = 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.

Value = 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.

Value = 4 (bit 2) Use the value passed in the `curtype` argument to control the cursor display. Options are shown below.

Value = 8 (bit 3) Don't display the error: console break message when the user presses Ctrl+C.

curtype If supplied, the *curtype* argument defines the type of cursor displayed. This must be an integer; options follow:

0 Display the normal crosshairs.

1 Don't display a cursor (no crosshairs).

2 Display the entity-selection "target" cursor.

Note: The *curtype* argument only affects the cursor type during the current `grread` function call.

Note: Be aware that additional control bits might be defined in future AutoCAD releases.

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 codes are as follows:

Table 4–6. Return values from grread

First element		Second element	
Value	**Type of input**	**Value**	**Meaning**
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, to 16001 to 16999	Screen menu box no. POP1 menu box no. POP2 menu box no. POP3 menu box no. … 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

Entering Ctrl+C while a `grread` is in progress aborts the AutoLISP program with a keyboard break (unless the *allkeys* argument has disallowed this).

Any other input is passed directly to **grread**, allowing 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 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 with an auxiliary function box. 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.

Note: Since input is handled differently on the various platforms supported by AutoCAD, the function **grread** may return unexpected results. See your *AutoCAD Interface, Installation, and Performance Guide* for more platform specific information.

- The default pointing device on platforms that use a system mouse return a code 11, not a code 6.
- On the Macintosh platform, the pop menus will return a code 11, not a code 4. Also on the MAC, a double click returns a code 11 (not a code 6), and is followed by a code 5 coordinate pair if selected in the current viewport. Conversely, a double click in a viewport other than the current one returns a code 3 coordinate pair followed by a code 11.

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

The **grtext** function writes into the text portions of the AutoCAD graphics screen. If called with a *box* number between 0 and the highest-numbered screen menu box minus 1, it displays the string argument *text* in the specified screen menu box.The *text* argument is truncated if it is too long to fit in the menu box, and it is filled with blanks if shorter. If an invalid box number is supplied, `nil` is returned.

If the optional integer argument *highlight* is supplied as a positive number, **grtext** highlights the text in the designated box.When writing to menu boxes, the text must first be written without the *highlight* argument, then it must be highlighted. 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.

This function displays just the supplied text in the menu area on screen; it does not change the underlying screen menu item.

If **grtext** is called with a box number of –1, it writes the text into the mode status line on screen. The length of the mode status line differs from display to display (most allow at least 40 characters). **grtext** truncates the text to fit in the available space.

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

Finally, **grtext** can be called with no arguments to restore all on-screen text areas to their standard values.

The SCREENBOXES system variable can be interrogated to determine the number of screen menu boxes present in a particular installation.

(grvecs *vlist [trans]*)

Draws multiple vectors on the graphics screen. The *vlist* argument is a list comprised of a series of optional colour integers and two point lists. The format for *vlist* is as follows:

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

The optional colour value applies to all succeeding vectors until *vlist* specifies another colour. The colour is specified as an integer. AutoCAD colours are in the range 0–255. If the colour value is greater than 255, succeeding vectors are drawn in *XOR ink*, which complements anything it draws over and erases itself when overdrawn. If the colour value is less than zero, the vector is highlighted.

Highlighting depends on the display device. Most display drivers indicate highlighting by a dashed line, but some indicate it by using a distinctive colour.

A pair of point lists specify the endpoints of the vectors, expressed in the current UCS; these can be two-dimensional or three-dimensional points.

Important: You must pass these points as pairs—that is, in two successive point lists—or the **grvecs** call will fail.

AutoCAD clips the vectors as required to fit on screen. If the call to **grvecs** is successful, it returns `nil`.

Examples

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

```
(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 optional *trans* argument is a transformation matrix that lets you 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. For example, the following matrix represents a uniform scale of 1.0 and a translation of 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)
)
```

If this matrix were applied to the above list of vectors, they would be offset by 5.0,5.0,0.0.

See also: The **nentselp** function on page 140 for more information on transformation matrixes.

(handent *handle*)

An entity's name can change from one editing session to the next, whereas an entity's handle remains constant throughout its life. Given an entity handle string as the *handle* argument, the **handent** function returns the entity name associated with that handle in the current editing session. Once the entity name has been obtained, it can be used to manipulate the entity with any of the entity-related functions.

Example

The following code,

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

in a particular editing session. Used with the same drawing but in another editing session, the same call might return a different entity name. The same entity is referenced in each case; its handle remains the same, but its entity name might change from session to session.

If handles are not being used in the drawing, or if **handent** is passed an invalid handle or a handle not used by any entity in the current drawing, `nil` is returned. The **handent** function will return entities that have been deleted during the current editing session; you can then undelete them, if you want, with the **entdel** function (see "(entdel ename)" on page 103).

(if *testexpr thenexpr [elseexpr]*)

This function conditionally evaluates expressions. If *testexpr* is not `nil`, then it evaluates *thenexpr*; otherwise it evaluates *elseexpr*. The last expression (*elseexpr*) is optional. The **if** function returns the value of the selected expression; if *elseexpr* is missing and *testexpr* is `nil`, then the **if** function returns `nil`.

Examples

```
(if (= 1 3) "YES!!" "no.")      returns    "no."
(if (= 2 (+ 1 1)) "YES!!")      returns    "YES!!"
(if (= 2 (+ 3 4)) "YES!!")      returns    nil
```

See also: The **progn** function on page 145.

(initget *[bits] [string]*)

This function establishes various options for use by the next **entsel**, **nentsel**, **nentselp**, or **get*xxx*** function (except **getstring**, **getenv**, and **getvar**). The **initget** function always returns `nil`.

The optional *bits* argument is an integer (bit-coded) with values as follows:

Table 4–7. Input options set by initget

Bit value	Meaning
1	Disallow null input
2	Disallow zero values
4	Disallow negative values
8	Do not check drawing limits, even if LIMCHECK is On
16	(Not currently used)
32	Use dashed lines when drawing rubber-band line or box
64	Disallow input of a *Z* coordinate (`getdist` only)
128	Returns arbitrary keyboard input

Caution: Future versions of AutoLISP might use additional **`initget`** control bits, so avoid setting bits that aren't shown in the table or described in this section.

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

Table 4–8. User-input functions and applicable control bits

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

The following list describes each control bit in more detail:

Value = 1 (bit 0) Prevents the user from responding to the request by entering only ↵.

Value = 2 (bit 1) Prevents the user from responding to the request by entering zero.

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

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.

Value = 16 (bit 4) Not currently used.

Value = 32 (bit 5) For those functions allowing the user to specify a point by selecting a location on the graphics screen, causes the rubber-band line or box displayed by the drawing editor to be dashed instead of solid (some display drivers use a distinctive colour instead of dashed lines). If the system variable POPUPS is zero, AutoCAD ignores this bit.

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

Value = 128 (bit 7) Allows arbitrary input as if it is a keyword, first honouring any other control bits and listed keywords. This bit takes precedence over bit 0; if bit 7 is set and the user enters ↵, a null string will be returned.

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

Keyword Specifications

The optional *`string`* argument defines a list of option keywords to be checked by the next `entsel`, `nentsel`, `nentselp`, or `getxxx` request if the user does not enter the expected type of input (for example, a point for `getpoint`). If the user input matches a keyword from this list, the supplied function returns that keyword as a string result. The user program can test for the keywords and perform the desired action for each one. If the user input is not of the expected type and does not match a keyword, AutoCAD asks the user to try again. A legal keyword can contain letters, numbers, or hyphens (-).

The *`string`* argument is interpreted according to these rules:

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

- Each keyword specification can instruct AutoCAD to recognize abbreviations. There are two methods of doing so:
    - The required portion of the keyword is specified in uppercase characters, and the remainder of the keyword is specified in lowercase. The uppercase abbreviation can be anywhere in the keyword (e.g., `"LType"`, `"eXit"`, or `"toP"`).
    - The *entire* keyword is specified in uppercase characters, and it is followed immediately by a comma, followed by a repetition of the required characters (e.g., `"LTYPE,LT"`). The keyword characters, in this case, must include the first letter of the keyword, which means that `"EXIT,X"` is not valid.

      (This second method is provided to assist the development of applications for languages that do not use uppercase and lowercase in the style of the Roman alphabet.)

  Both of the brief examples, `"LType"` and `"LTYPE,LT"`, are equivalent: if the user types **LT** (in either uppercase or lowercase), 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, and something like **LTSCALE** or **LTYPEX** would not match the spelling of the keyword.

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

Important: The control flags and keyword list established by **`initget`** are applied only to the next **`entsel`**, **`nentsel`**, **`nentselp`**, or **`getxxx`** call and are then automatically discarded. This avoids the need for another function call to clear the special conditions.

See also: "Control of User-input Function Conditions" on page 31.

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

The **`inters`** function examines two lines and returns the point where they intersect, or `nil` if they do not intersect. The *`pt1`* and *`pt2`* arguments are the endpoints of the first line, and *`pt3`* and *`pt4`* are the endpoints of the second line.

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

If the optional *`onseg`* argument is present and is `nil`, the lines defined by the four *`pt`* arguments are considered infinite in length, and **`inters`** returns the point where they 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`.

For example, given:

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

then

```
(inters a b c d)        returns   nil
(inters a b c d T)      returns   nil
(inters a b c d nil)    returns   (4.0 4.0)
```

(itoa *int*)

This function returns the conversion of an integer, *int*, into a string.

Examples

```
(itoa 33)               returns   "33"
(itoa -17)              returns   "-17"
```

(lambda *arguments expr* ...)

The `lambda` function defines an *anonymous* function. This is typically used 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. For example:

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

and

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

(last *list*)

This function returns the last element in *list*. The *list* argument must not be `nil`. For example:

```
(last '(a b c d e))        returns   E
(last '(a b c (d e)))      returns   (D E)
```

As shown, `last` can return an atom or a list.

Note: At first glance, **last** might seem like a perfect way to obtain the *Y* coordinate of a point. While this is true for 2D points (lists of two reals), **last** would return the *Z* coordinate of a 3D point. To let your functions work properly given either 2D or 3D points, we suggest you use **cadr** to obtain *Y* coordinates and **caddr** to obtain *Z* coordinates.

(length *list*)

This function returns an integer indicating the number of elements in `list`. For example:

```
(length '(a b c d))        returns   4
(length '(a b (c d)))      returns   3
(length '())               returns   0
```

(list *expr* ...)

This function takes any number of expressions (`expr`) and strings them together, returning a list. For example:

```
(list 'a 'b 'c)            returns   (A B C)
(list 'a '(b c) 'd)        returns   (A (B C) D)
(list 3.9 6.7)             returns   (3.9 6.7)
```

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

Note: As an alternative to using the **list** function, you can explicitly quote a list if there are no variables or undefined items in the list.

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

This can be quite handy for creating association lists and defining points. See **quote** on page 146.

(listp *item*)

This function returns `T` if `item` is a list, and `nil` otherwise. For example:

```
(listp '(a b c))    returns    T
(listp 'a)          returns    nil
(listp 4.343)       returns    nil
(listp nil)         returns    T       nil is both an atom and a list
```

(load *filename [onfailure]*)

This function loads a file of AutoLISP expressions and evaluates those expressions. The `filename` argument is a string that represents the filename without an extension (an extension of *.lsp* is assumed). The `filename` can include a directory prefix, as in *"/function/test1"*. On DOS systems, a drive letter is also permitted, and you can use the backslash instead of the forward slash (but remember that you must use \\ to obtain one backslash in a string).

If you don't include a directory prefix in the *filename* string, **load** searches the AutoCAD library path for the specified file, in a manner similar to that of the **findfile** function (the AutoCAD library path is described on page 114). If the file is found anywhere on this path, **load** then loads the file.

If the operation is successful, **load** returns the value of the last expression in the file, which is often the name of the last function defined in the file. If the **load** operation fails, it normally causes an AutoLISP error. However, if the *onfailure* argument is supplied, **load** returns the value of this argument upon failure instead of issuing an error message. This allows an AutoLISP application calling **load** to take alternative action upon failure.

You should ensure that the *onfailure* argument is different from the last expression in the file; otherwise, the meaning of the value returned by **load** is ambiguous. Note that if the *onfailure* argument is a valid AutoLISP function, it is evaluated. Therefore, the *onfailure* argument should be a string or an atom in most cases.

For example, assuming that file */fred/test1.lsp* contains

```
(defun MY-FUNC1 (x)
    ...function body...
)
(defun MY-FUNC2 (x)
    ...function body...
)
```

and that file *test2.lsp* does not exist, then

```
(load "/fred/test1")          returns   MY-FUNC2
(load "\\fred\\test1")        returns   MY-FUNC2
(load "/fred/test1" "bad")    returns   MY-FUNC2
(load "test2" "bad")          returns   "bad"
(load "test2")                causes an AutoLISP error
```

The **load** function can be used from within another AutoLISP function, or even recursively (in the file being loaded).

Each time an AutoCAD drawing editor session begins, AutoLISP loads the file *acad.lsp* if it exists. You can place function definitions in this file, and they are automatically evaluated (defined) each time you begin editing a drawing. If you want to have a series of AutoCAD commands or AutoLISP functions executed automatically when beginning, saving, or exiting a drawing, place a **defun** of the special **S::STARTUP** function in the *acad.lsp* file; if this function exists, AutoCAD executes the *acad.lsp* file automatically. Chapter 8 of the *AutoCAD Customization Manual* discusses *acad.lsp* and the use of the **load** function.

See also: The **defun** function on page 101 and "Defining Functions and Automatic Loading" on page 14.

(log *number*)

This function returns the natural log of *number* as a real. For example:

```
(log 4.5)      returns   1.50408
(log 1.22)     returns   0.198851
```

(logand *number number ...*)

This function returns the result of a logical bitwise AND of a list of *numbers*. The *numbers* must be integers, and the result is also an integer. For example:

```
(logand 7 15 3)          returns    3
(logand 2 3 15)          returns    2
(logand 8 3 4)           returns    0
```

(logior *integer ...*)

This function returns the result of a logical bitwise inclusive OR of a list of *numbers*. The *numbers* must be integers, and the result is also an integer. For example:

```
(logior 1 2 4)           returns    7
(logior 9 3)             returns    11
```

(lsh *num1 numbits*)

This function returns the logical bitwise shift of *num1* by *numbits* bits. The *num1* and *numbits* arguments must be integers, and the result is also an integer.

If *numbits* is positive, *num1* is shifted to the left; if negative, to the right. In either case, *zero* bits are shifted in, and the bits shifted out are discarded. If a *one* bit is shifted into or out of the top bit (16th on 16-bit machines, 32d on 32-bit workstations) of an integer, its sign changes.

For example:

```
(lsh 2 1)                returns    4
(lsh 2 -1)               returns    1
(lsh 40 2)               returns    160
```

Note: The call (lsh 16384 1) returns –32,768 on 16-bit machines, but it returns 32,768 on 32-bit machines.

(mapcar *function list1 ... listn*)

The **mapcar** function returns a list as the result of executing *function* with the individual elements of *list1* through *listn* supplied as arguments to *function*. The number of *lists* must match the number of arguments required by *function*. For example:

```
(setq a 10 b 20 c 30)
(mapcar '1+ (list a b c))                 returns    (11 21 31)
```

is equivalent to

```
(1+ a)
(1+ b)
(1+ c)
```

except that **mapcar** returns a list of the results. Likewise

```
(mapcar '+ '(10 20 30) '(4 3 2))     returns     (14 23 32)
```

This is similar to

```
(+ 10 4)
(+ 20 3)
(+ 30 2)
```

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. For instance:

```
(mapcar '(lambda (x)
           (+ x 3)
         )
         '(10 20 30)
)
```

returns

```
(13 23 33)
```

and

```
(mapcar  '(lambda (x y z)
           (* x (- y z))
          )
          '(5 6) '(20 30) '(14 5.0)
)
```

returns

```
 (30 150.0)
```

(max *number number* ...)

This function returns the largest of the *numbers* given. Each *number* can be a real or an integer. If all the *numbers* are integers, the result is an integer; if any of the *numbers* are real numbers, the integers are promoted to real numbers and the result is a real number.

For example:

```
(max 4.07 -144)       returns     4.07
(max -88 19 5 2)      returns     19
(max 2.1 4 8)         returns     8.0
```

(mem)

Displays the current state of AutoLISP's memory. See "Memory Statistics" on page 178 for more information on **mem**.

(member *expr list*)

This function searches `list` for an occurrence of *expr* and returns the remainder of `list` starting with the first occurrence of *expr*. If there is no occurrence of *expr* in `list`, **member** returns `nil`. For example:

```
(member 'c '(a b c d e))    returns    (C D E)
(member 'q '(a b c d e))    returns    nil
```

(menucmd *string*)

The **menucmd** function lets AutoLISP programs switch between subpages in an AutoCAD menu. Thus, an AutoLISP program can work in concert with an associated menu file, displaying an appropriate submenu of alternatives whenever user input is needed. The **menucmd** function always returns `nil`. The *string* argument is of the form:

"*section=submenu*"

where *section* specifies the menu section and *submenu* specifies which submenu to activate within that section. The allowed values of *section* are the same as they are in menu file submenu references; these values are shown in the following table:

Table 4–9. Section string values

Section string	Menu section
S	For the SCREEN menu
B1-B4	For BUTTONS menus 1 through 4
I	For the ICON menu
P0–P16	For pull-down (POP) menus 0 through 16
T1–T4	For TABLET menus 1 through 4
A1-A4	For AUX menus 1 through 4
M	For DIESEL string expressions

Note: For compatibility with previous AutoLISP versions, a `"B"` supplied as the section string will be interpreted as a `"B1"`.

See also: Chapter 6 of the *AutoCAD Customization Manual* and "Interactive Output" on page 38 for more information and examples.

(min *number number ...*)

This function returns the smallest of the *numbers* given. Each *number* can be a real or an integer. If all the *numbers* are integers, the result is an integer; if any of the *numbers* are real numbers, the integers are promoted to real numbers and the result is a real number. For example:

```
(min 683 -10.0)       returns    -10.0
(min 73 2 48 5)       returns    2
(min 2 4 6.7)         returns    2.0
```

(minusp *item*)

This function returns `T` if *`item`* is a real or integer and evaluates to a negative value; otherwise it returns `nil`. It is not defined for other types of *`item`*. For example:

```
(minusp -1)           returns     T
(minusp -4.293)       returns     T
(minusp 830.2)        returns     nil
```

(nentsel *[prompt]*)

This function provides access to entity definition data contained within an insert entity (inside a Block).

The **nentsel** function prompts the user to select an object. The current setting of Osnap is ignored (no object snap) unless the user specifically requests it . To provide additional support at the Command: prompt, **nentsel** can optionally honour keywords defined by a previous call to **initget**.

The optional *`prompt`* argument, if provided, must be a string. If it is omitted, the standard Select objects: prompt will be issued.

When the selected object is *not* a complex entity (a Polyline or Block), **nentsel** returns the same information as **entsel**. However, if the selected entity is a 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 Polyline segment. Picking the third segment of a Polyline, for example, returns the third vertex. The Seqend subentity is never returned by **nentsel** for a Polyline.

When the selected object is a component of a Block, **nentsel** returns a list containing four elements as described next. The exception to this is Attributes within a Block. Selecting an Attribute within a Block returns only the name of the Attribute and the pick point (similar to the list returned by **entsel**).

The first element of the list returned from picking an entity 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 entity.

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.

The fourth element is a list containing the entity name of the Block that contains the selected entity. If the selected entity is contained in a nested Block (a Block within a Block), the list additionally contains the entity names of all Blocks in which the selected entity is nested, starting with the innermost Block and continuing outward until the name of the Block that was inserted in the drawing is reported.

```
(<Entity Name: ename1>           Name of entity
 (Px Py Pz)                      Pick point
 ( (X0 Y0 Z0)                    Model to World
   (X1 Y1 Z1)                    Transformation Matrix
   (X2 Y2 Z2)
   (X3 Y3 Z3)
 )
  (<Entity name: ename2>         Name of most deeply nested Block
          .                      containing selected entity
          .
          .
  <Entity name: enamen>)         Name of outermost Block
)                                containing selected entity
```

Once the entity name and the Model to World Transformation Matrix are obtained, you can transform the entity definition data points from the MCS to the WCS. Use **entget** and **assoc** on the entity name to obtain the desired definition points expressed in MCS coordinates. The Model to World Transformation Matrix returned by **nentsel** has the same purpose as that returned by **nentselp**, but it is a 4×3 matrix—passed as an array of four points—that uses the convention that a point is a row rather than a column. The transformation is described by the following matrix multiplication:

$$\begin{bmatrix} X' & Y' & Z' & 1.0 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1.0 \end{bmatrix} \cdot \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \\ M_{30} & M_{31} & M_{32} \end{bmatrix}$$

So the equations for deriving the new coordinates are as follows:

$$X' = XM_{00} + YM_{10} + ZM_{20} + M_{30}$$
$$Y' = XM_{01} + YM_{11} + ZM_{21} + M_{31}$$
$$Z' = XM_{02} + YM_{12} + ZM_{22} + M_{32}$$

The M_{ij}, where $0 \le i, j \le 2$, are the Model to World Transformation Matrix coordinates, X, Y, and Z is the entity definition data point expressed in MCS coordinates, and X', Y', and Z' is the resulting entity definition data point expressed in WCS coordinates.

Note: This 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 ADS functions.

See also: "Entity Name Functions" on page 52, **entsel** on page 110, and **initget** on page 128.

(nentselp *[prompt] [pt]*)

The **nentselp** function provides similar access to entity definition data contained within a Block to that provided by the **nentsel** function. In addition to the optional *prompt* argument this function also accepts a selection point to be supplied as an optional argument: this allows entity selection without user input. The **nentselp** function returns a 4×4 transformation matrix, defined as follows:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{32} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{bmatrix}$$

Where the first three columns of the matrix specify scaling and rotation. The fourth column of the matrix is a translation vector.

The functions that use a matrix of this type use the convention of treating 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 $\begin{bmatrix} M_{30} & M_{31} & M_{32} M_{33} \end{bmatrix}$, has the nominal value of $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$; 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:

$$X' = XM_{00} + YM_{01} + ZM_{02} + M_{03}\,(1.0)$$

$$Y' = XM_{10} + YM_{11} + ZM_{12} + M_{13}\,(1.0)$$

$$Z' = XM_{20} + YM_{21} + ZM_{22} + M_{23}\,(1.0)$$

As these equations show, the scale factor and the last row of the matrix have no effect and are ignored.

See also: "Transformation Matrixes" in chapter 1 of the *AutoCAD Development System Programmer's Reference* for more information.

(not *item*)

This function returns `T` if the expression *`item`* is `nil`, and `nil` otherwise. Typically, the **`null`** function is used for lists, and **`not`** is used for other data types in conjunction with some type of control function. For example, given the following assignments:

```
(setq a 123)
(setq b "string")
(setq c nil)
```

then

```
(not a)          returns     nil
(not b)          returns     nil
(not c)          returns     T
(not '())        returns     T
```

(nth *n list*)

This function returns the *nth* element of `list`, where *`n`* is the number of the element to return (zero is the first element). If *`n`* is greater than `list`'s highest element number, it returns `nil`. For example:

```
(nth 3 '(a b c d e))   returns    D
(nth 0 '(a b c d e))   returns    A
(nth 5 '(a b c d e))   returns    nil
```

(null *item*)

This function returns `T` if *`item`* is bound to `nil`, and `nil` otherwise. For example, given the following assignments:

```
(setq a 123)
(setq b "string")
(setq c nil)
```

then

```
(null a)         returns     nil
(null b)         returns     nil
(null c)         returns     T
(null '())       returns     T
```

(numberp *item*)

This function returns `T` if *`item`* is a real or an integer, and `nil` otherwise. For example, given the assignments:

```
(setq a 123)
(setq b 'a)
```

then

```
(numberp 4)                 returns        T
(numberp 3.8348)            returns        T
(numberp "Howdy")           returns        nil
(numberp 'a)                returns        nil
(numberp a)                 returns        T
(numberp b)                 returns        nil
(numberp (eval b))          returns        T
```

(open *filename mode*)

This function opens a file for access by the AutoLISP I/O functions. It returns a file descriptor to be used by the other I/O functions; therefore it must be assigned to a symbol using the **setq** function. For example:

```
(setq a (open "file.ext" "r"))
```

The *`filename`* argument is a string specifying the name and extension of the file to be opened. The *`mode`* argument is the read/write flag. It must be a string containing a single lowercase letter. The following table describes the valid mode letters:

Table 4–10. Mode options for the open function

Open mode	Description
"r"	Open for reading. If *`filename`* does not exist, it returns `nil`.
"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.
"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. ***Caution:*** 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 more data is present after it. If you intend to use OPEN's "a" mode to append data to files produced by another program, ensure that the other program does not insert CTRL Z markers at the end of its text files.

Assuming that the files named in the following examples do *not* exist:

```
(setq f (open "new.tst" "w"))          returns     <File #nnn>
(setq f (open "nosuch.fil" "r"))       returns     nil
(setq f (open "logfile" "a"))          returns     <File #nnn>
```

The `filename` argument can include a directory prefix, as in */test/func3*. On DOS systems, a drive letter is also permitted, and you can use the backslash instead of the forward slash (but remember that you must use \\ to obtain one backslash in a string).

For example:

```
(setq f (open "/x/new.tst" "w"))     returns     <File #nnn>
(setq f (open "nosuch.fil" "r"))     returns     nil
```

(or *expr* ...)

This function returns the logical OR of a list of expressions. The **or** function evaluates the expressions from left to right, looking for a non-`nil` expression. If one is found, **or** ceases further evaluation and returns `T`. If all of the expressions are `nil`, **or** returns `nil`. For example:

```
(or nil 45 '())     returns     T
(or nil '())        returns     nil
```

(osnap *pt mode-string*)

This function returns a 3D point that is the result of applying the object snap modes described by `mode-string` to point `pt`. The `mode-string` argument is a string consisting of one or more valid object snap identifiers such as `midpoint`, `center`, and so on, separated by commas. For instance:

```
(setq pt2 (osnap pt1 "cen"))
```

Operation of this function is dependent on the current 3D view. See the *AutoCAD Reference Manual* for details.

See also: "Object Snap" on page 25.

pi

This is not a function, but rather the constant π. It evaluates to approximately 3.1415926.

(polar *pt angle distance*)

This function returns the UCS 3D point at angle `angle` and distance `distance` from UCS point `pt`. The `angle` argument is expressed in radians from the *X* axis, with angles increasing in the counterclockwise direction. Although `pt` can be a 3D point, `angle` is always with respect to the current construction plane. For example:

```
(polar '(1 1 3.5) 0.785398 1.414214)     returns     (2.0 2.0 3.5)
```

(prin1 *[expr [file-desc]]*)

This function prints the expression *expr* on screen and returns *expr*. The *expr* argument can be any expression; it need not be a string. If *file-desc* is present (and is a file descriptor for a file opened for writing), *expr* is written to the file exactly as it would appear on screen. Only the specified *expr* is printed; no newline or space is included. For example, given the assignment:

```
(setq a 123)
(setq b '(a))
```

then

```
(prin1 'a)          prints    A          and returns    A
(prin1 a)           prints    123        and returns    123
(prin1 b)           prints    (A)        and returns    (A)
(prin1 "Hello")     prints    "Hello"    and returns    "Hello"
```

Each of the preceding examples prints on screen, since no *file-desc* was specified. Assuming that `f` is a valid file-descriptor for a file opened for writing:

```
(prin1 "Hello" f)
```

would write `"Hello"` to the specified file and return `"Hello"`.

If *expr* is a string containing control characters, **prin1** expands these characters with a leading \, as follows:

Table 4–11. Control codes

Code	Meaning
`\\`	\ character
`\"`	" character
`\e`	Escape character
`\n`	Newline character
`\r`	Return character
`\t`	Tab character
`\`*nnn*	Character whose octal code is *nnn*

Thus

```
(prin1 (chr 2))     prints    "\002"    and returns    "\002"
(prin1 (chr 10))    prints    "\n"      and returns    "\n"
```

The **prin1** function can be called with no arguments, in which case it returns (and prints) the null string. If you use **prin1** (with no arguments) as the last expression in a user-defined function, only a blank line is printed when the function is complete, letting the user exit "quietly" from a function. For example, given

```
    (defun C:SETUP ()
        (setvar "LUNITS" 4)
        (setvar "BLIPMODE" 0)
        (prin1)
    )
```

then

```
    Command: setup
```

executes the user-defined command, performs the requested **setvar** functions, and returns to the Command: prompt without printing any extraneous messages.

See also: "Interactive Output" on page 38.

(princ *[expr [file-desc]]*)

This function is the same as **prin1**, except that 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**.

See also: "Interactive Output" on page 38.

(print *[expr [file-desc]]*)

This function is the same as **prin1**, except that it prints a newline character before *expr* and prints a space following *expr*.

See also: "Interactive Output" on page 38.

(progn *expr* ...)

This function evaluates each *expr* sequentially and returns the value of the last expression. You can use **progn** to evaluate several expressions where only one expression is expected.

Example

```
    (if (= a b)
        (progn
            (setq a (+ a 10))
            (setq b (- b 10))
        )
    )
```

The **if** function normally evaluates one *then* expression if the test expression evaluates to anything but nil. In this example, we have used **progn** to cause two expressions to be evaluated instead.

(prompt *msg*)

This function displays `msg` on your screen's prompt area and returns `nil`. The `msg` argument is a string. On dual-screen AutoCAD configurations, **`prompt`** displays `msg` on both screens and is therefore preferable to **`princ`**.

Example

`(prompt "New value: "`	displays	New value:	on the screen(s)

and returns `nil`.

See also: "Interactive Output" on page 38.

(quit)

The **`quit`** function forces the current application to 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 on page 113.

(quote *expr*)

Returns `expr` unevaluated. This can also be written

```
'expr
```

For example:

`(quote a)`	returns	`A`
`(quote cat)`	returns	`CAT`
`(quote (a b))`	returns	`(A B)`
`'a`	returns	`A`
`'cat`	returns	`CAT`
`'(a b)`	returns	`(A B)`

The last three examples don't work if entered directly from the keyboard in response to an AutoCAD prompt. Remember that such input must begin with a " **(** " or " **!** " character to be recognized as a LISP expression.

(read *string*)

This function returns the first list or atom obtained from `string`. The `string` argument cannot contain blanks except within a list or string. The **`read`** function returns its argument converted into the corresponding data type, as shown in the following examples:

`(read "hello")`	returns the atom	`HELLO`
`(read "hello there")`	returns the string	`HELLO`
`(read "\"Hi Y'all\"")`	returns the string	`"Hi Y'all"`
`(read "(a b c)")`	returns the list	`(A B C)`
`(read "(a b c) (d)")`	returns the list	`(A B C)`
`(read "1.2300")`	returns the real number	`1.23`
`(read "87")`	returns the integer	`87`
`(read "87 3.2")`	returns the integer	`87`

(read-char *[file-desc]*)

This function reads a single character from the keyboard input buffer or from the open file described by *`file-desc`*. It returns the (integer) decimal ASCII code representing the character read (see appendix F for a list of ASCII codes).

If no *`file-desc`* is specified and there are no characters in the keyboard input buffer, **`read-char`** waits for you to enter something at the keyboard (followed by ↵). For instance, assuming the keyboard input buffer is empty,

```
(read-char)
```

waits for something to be entered. If you enter ABC followed by ↵, **`read-char`** returns 65 (the decimal ASCII code for the letter A). The next three calls to **`read-char`** return 66, 67, and 10 (newline), respectively. If another **`read-char`** call is then made, it again waits for input.

The various operating systems under which AutoCAD and AutoLISP run use several different conventions to signal the end of a line in an ASCII text file. UNIX systems, for example, uses a single newline character (linefeed (LF), ASCII code 10), whereas DOS systems use a pair of characters (carriage-return (CR)/LF, ASCII codes 13 and 10) for the same purpose. To facilitate development of AutoLISP programs that work in a portable fashion on all supported operating systems, **`read-char`** accepts all these conventions, returning a single newline character (ASCII code 10) whenever it detects an end-of-line character (or character sequence).

(read-line *[file-desc]*)

This function reads a string from the keyboard or from the open file described by *`file-desc`*. If **`read-line`** encounters the end of the file, it returns `nil`; otherwise, it returns the string that it read. For example, assuming `f` is a valid open file pointer:

```
(read-line f)
```

returns the next input line from the file, or `nil` if the end-of-file has been reached.

(redraw *[ename [mode]]*)

The effect of the function depends on the number of arguments supplied. If called with no arguments:

```
(redraw)
```

it redraws the current viewport, just like the AutoCAD REDRAW command. If called with an entity name argument:

```
(redraw ename)
```

it redraws the specified entity. This is useful in identifying an entity on the screen after using **`grclear`** to clear the screen. See "Entity Name Functions" on page 52 for descriptions of entity names.

Complete control over the redrawing of an entity is provided by calling **redraw** with two arguments:

```
(redraw ename mode)
```

where *ename* is the entity name to be redrawn and *mode* is an integer with one of the following values:

Table 4–12. Modes for redraw

Redraw mode	Action
1	Redraw entity
2	Undraw entity (blank it out)
3	Highlight entity
4	Unhighlight entity

If *ename* is the header of a complex entity (a Polyline or a Block with attributes), it 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.

The **redraw** function always returns `nil`.

(regapp *application*)

This function registers an application name with the current AutoCAD drawing. An application name is the principal mechanism for grouping, storing, retrieving, and modifying application-defined extended entity data. An application can use as many application names as desired to organize extended entity data.

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 entity data in the drawing. This allows an application to distinguish its extended entity data from that of other applications. Applications that append or modify extended entity data can be either AutoLISP applications or AutoCAD Development System (ADS) applications.

The *application* argument is a string up to 31 characters long that adheres to the symbol naming conventions (such as table names). An application name can contain letters, digits, and the special characters $ (dollar sign), - (hyphen), and _ (underscore). It *cannot* contain spaces. Letters in the name are converted to uppercase.

Examples

```
(regapp "ADESK_4153322344")
(regapp "DESIGNER-v2.1-124753")
```

Note: We recommend you pick an application name that is guaranteed to be unique. 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 could be included in the application name or stored by the application in a separate integer or real number field; for example: `(1040 2.1)`. Examples of such schemes are shown above.

See also: The *AutoCAD Reference Manual* for information on extended entity data and the *AutoCAD Development System Programmer's Reference* for information on ADS.

(rem *num1 num2 ...*)

This function divides *num1* by *num2* and returns the remainder (*num1* mod *num2*). The **rem** function can be used with reals or integers, with standard rules of promotion. For example:

```
(rem 42 12)            returns    6
(rem 12.0 16)          returns    12.0
(rem 60 3)             returns    0
```

(repeat *number expr ...*)

In this function, *number* represents any positive integer. The function evaluates each *expr number* times and returns the value of the last expression. For example, given the assignments:

```
(setq a 10)
(setq b 100)
```

then

```
(repeat 4
    (setq a (+ a 10))
    (setq b (+ b 100))
)
```

sets a *to 50, sets* b *to 500, and returns 500*

(reverse *list*)

This function returns *list* with its elements reversed. For example:

```
(reverse '((a) b c))    returns    (C B (A))
```

(rtos *number [mode [precision]]*)

This function returns a string that is the representation of *number* (a real) according to the settings of *mode*, *precision*, the AutoCAD UNITMODE system variable, and the DIMZIN dimensioning variable. The *mode* and *precision* arguments are integers that select the linear units mode and precision. The supported *mode* values are listed next:

Table 4–13. Linear units values

Mode value	String format
1	Scientific
2	Decimal
3	Engineering (feet and decimal inches)
4	Architectural (feet and fractional inches)
5	Fractional

The *mode* and *precision* arguments correspond to the AutoCAD system variables LUNITS and LUPREC. If you omit the arguments, **rtos** uses the current settings of LUNITS and LUPREC.

The UNITMODE variable affects the returned string when engineering, architectural, or fractional units are selected (a *mode* value of 3, 4, or 5).

See also: The section titled "String Conversions" on page 33, which continues the discussion of **rtos**.

(set *sym expr*)

This function sets the value of *sym* (where *sym* is a quoted symbol name) to *expr*, and returns that value. For example:

```
(set 'a 5.0)          returns    5.0    and sets symbol   A
(set (quote b) 'a)    returns    A      and sets symbol   B
```

If **set** is used with an unquoted symbol name, it can assign a new value to another symbol *indirectly*. For instance, given the previous examples:

```
(set b 640)           returns    640
```

and assigns the value 640 to symbol a (since that's what symbol b contains).

Related topics: See **setq**, described in the following section.

(setq *sym1 expr1 [sym2 expr2]* ...)

This function sets the value of *sym1* to *expr1*, *sym2* to *expr2*, and so on. This is the basic assignment function in AutoLISP. The **setq** function can assign multiple symbols in one call to the function, but returns only the last *expr*. For example:

```
(setq a 5.0)          returns    5.0
```

and sets the symbol a to 5.0. Whenever a is evaluated, it evaluates the real number 5.0. These are other examples:

```
(setq b 123 c 4.7)    returns    4.7
(setq s "it")         returns    "it"
(setq x '(a b))       returns    (A B)
```

There is a maximum length of 132 characters for strings assigned to a symbol directly by either **setq** or **set**. However, you can create longer strings by using the **strcat** function to join several strings together and then assign the result to a symbol.

The **setq** function is the same as the **set** function except that the symbol name is not quoted. In other words, **set** evaluates its first argument, but **setq** does not. This example shows the similarity between these two functions.

```
(setq a 5.0)    is equivalent to    (set (quote a) 5.0)
```

The **set** and **setq** functions create or modify global symbols, unless used within a **defun** to assign a value to a function argument or to a symbol declared as local to that **defun**. For instance:

```
(setq glo1 123)                         Creates a global symbol
(defun demo (arg1 arg2 / loc1 loc2)
    (setq arg1 234)                     Assigns a new value locally
    (setq loc1 345)                     Assigns a new value locally
    (setq glo1 456)                     Assigns a new value globally
    (setq glo2 567)                     Creates a new global symbol
)
```

Global symbols can be accessed or modified by any function or used in any expression. Local symbols and function arguments are meaningful only during evaluation of the function that defines them (and during functions called by that function). Function arguments are treated as local symbols: the function can change their values, but the changes are discarded when the function exits.

Warning: The **set** and **setq** functions can assign new values to the AutoLISP built-in symbols and function names, thereby discarding the original assignments or making them inaccessible. Several users have been unfortunate enough to try such things as these:

```
(setq angle (...))        Wrong!
(setq length (...))       Wrong!
(setq max (...))          Wrong!
(setq t (...))            Wrong!
(setq pi 3.0)             Wrong!!!
```

To avoid strange errors, be careful when choosing names for your own symbols. *Never use a built-in symbol or function name for your own symbol.* If you are unsure of the originality of a symbol name, you can enter the following at the prompt line (let's assume you want to check for the symbol mysym):

```
Command: (atoms-family 0 '("mysym"))
```

This would return

```
(nil)
```

if this symbol is not currently defined. See the **atoms-family** function page 95 for more information on the use of this function.

(setvar *varname value*)

This function sets an AutoCAD system variable `varname` to the given `value`, and returns that value. You must enclose the variable name in double quotes. For example:

```
(setvar "FILLETRAD" 0.50)      returns    0.5
```

sets the AutoCAD fillet radius to 0.5 units. For system variables with integer values, the supplied `value` must be between –32,768 and +32,767.

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.

Note: 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.

Caution: 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 appendix A of the *AutoCAD Reference Manual*.

Related topics: See **`getvar`** on page 123.

(sin *angle*)

This function returns the sine of `angle` as a real, where `angle` is expressed in radians. For example:

```
(sin 1.0)       returns    0.841471
(sin 0.0)       returns    0.0
```

(sqrt *number*)

This function returns the square root of `number` as a real. For example:

```
(sqrt 4)        returns    2.0
(sqrt 2.0)      returns    1.41421
```

(ssadd *[ename [ss]]*)

If called with no arguments, **`ssadd`** constructs a new selection set with no members. If called with a single entity name argument `ename`, **`ssadd`** constructs a new selection set containing that single entity. If called with an entity name and a selection set `ss`, **`ssadd`** adds the named entity to the selection set. The **`ssadd`** function always returns the new or modified selection set. 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; no error is reported. These are examples:

`(setq e1 (entnext))`	*Sets* `e1` *to name of first entity in drawing*
`(setq ss (ssadd))`	*Sets* `ss` *to a null selection set*
`(ssadd e1 ss)`	*Returns selection set* `ss` *with entity name* `e1` *added*
`(setq e2 (entnext e1))`	*Gets entity following* `e1`
`(ssadd e2 ss)`	*Returns selection set* `ss` *with entity name* `e2` *added*

(ssdel *ename ss*)

The **ssdel** function deletes the entity *ename* from selection set *ss* and returns the name of selection set *ss*. Note that the entity is actually deleted from the selection set as opposed to a new set being returned with the element deleted. If the entity is not in the set, `nil` is returned.

For example, given that entity name `e1` is a member of selection set `ss1` and entity name `e2` is not, then

`(ssdel e1 ss1)`	*Returns selection set* `ss` *with entity* `e1` *removed*
`(ssdel e2 ss1)`	*Returns* `nil` *(does not change* `ss`*)*

(ssget *[mode] [pt1 [pt2]] [pt-list] [filter-list]*)

The **ssget** function returns a selection set. The optional *mode* argument is a string specifying the entity selection method, which can be `"W"`, `"WP"`, `"C"`, `"CP"`, `"L"`, `"P"`, `"I"`, or `"F"`, corresponding to the AutoCAD Window, WPolygon Crossing, CPolygon, Last, Previous, Implied, and Fence selection modes. Another optional *mode* value is `"X"`, which selects the entire database. The *pt1* and *pt2* arguments specify points relevant to the selection. Supplying a point with no *mode* argument is equivalent to entity selection by picking a single point. The current setting of the Osnap command is ignored by this function (no object snap) unless you specifically request it while you are in the function. The *filter-list* argument may be used with any of the modes and allows finer control over the entity selection process.

If all arguments are omitted, **ssget** prompts the user through the AutoCAD general Select objects: mechanism, allowing interactive construction of the selection set.

Selection sets can contain entities from both paper and model space, but when the selection set is used in an operation, entities from the space not currently in effect are filtered out. This is also true for all AutoCAD commands.

Selection sets returned by **ssget** contain main entities only (no Attributes or Polyline vertices). These are examples of **ssget**:

`(ssget)`	*Asks the user for a general entity selection and places those entities in a selection set (sel. set)*
`(ssget "P")`	*Creates a sel. set of the most recently selected objects*
`(ssget "L")`	*Creates a sel. set of the last visible entity added to the database*
`(ssget "I")`	*Creates a sel. set of the entities in the Implied selection set (those selected while PICKFIRST is in effect)*

`(ssget '(2 2))`	*Creates a sel. set of the entity passing through (2,2)*
`(ssget "W" '(0 0) '(5 5))`	*Creates a sel. set of the entities inside the window from (0,0) to (5,5)*
`(ssget "C" '(0 0) '(1 1))`	*Creates a sel. set of the entities crossing the box from (0,0) to (1,1)*
`(ssget "X")`	*Creates a sel. set of all entities in the database*
`(ssget "X" filter-list)`	*Scans the database and creates a sel. set of entities matching the filter-list*
`(ssget filter-list)`	*Asks the user for a general entity selection and places only those entities matching the filter-list in a sel. set*
`(ssget "P" filter-list)`	*Creates a sel. set of the most recently selected entities that match the filter-list*

The following examples of **ssget** require that a list of points be passed to the function.

```
(setq pt_list '((1 1)(3 1)(5 2)(2 4)) )
```

`(ssget "WP" pt_list)`	*Creates a sel. set of all entities inside the polygon defined by* `pt_list`
`(ssget "CP" pt_list)`	*Creates a sel. set of all entities crossing and inside the polygon defined by* `pt_list`
`(ssget "F" pt_list)`	*Creates a sel. set of all entities intersecting the fence defined by* `pt_list`
`(ssget "WP" pt_list filter-list)`	*Creates a sel. set of all entities inside the polygon defined by* `pt_list` *that match the filter-list*

The selected objects are highlighted only when **ssget** is used with no arguments. No information about how the entity was picked is retained (see **entsel** for an alternative). Selection sets consume AutoCAD temporary file slots, so AutoLISP is not permitted to have more than 128 open at once. If this limit is reached, AutoCAD refuses to create any more selection sets and returns `nil` to all **ssget** calls. To close an unneeded 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.

Selection Set Filters

Selection set filter lists can be used with any of the selection modes. A *filter-list* is an association list, similar to the type of list returned by the **entget** function. The *filter-list* specifies which property (or properties) of the entities are to be checked, and what values constitute a match.

Using this mechanism, you can obtain a selection set including all entities of a given type, on a given layer, or of a given colour. The following example returns a selection set consisting only of blue lines that are part of the Implied selection set (those entities selected while PICKFIRST is in effect):

```
(ssget "I" '((0 . "LINE") (62 . 5)))
```

Using the **ssget** filter list, you can select all entities containing extended entity data (Xdata) for a particular application. This is done using the –3 group code, as in

```
(ssget "P" '((0 . "CIRCLE") (-3 ("APPNAME"))))
```

This selects all circles containing Xdata for the `"APPNAME"` application.

See also: "Selection Set Filter Lists" on page 47 and "Filtering for Extended Entity Data" on page 49.

Relational Tests

Unless otherwise specified, an "equals" test is implied for each item in the *filter-list*. For numeric groups (integers, reals, points, and vectors) you can specify other relations by including a special –4 group code that specifies a relational operator. The value of a –4 group is a string indicating the test operator to be applied to the next group in the filter list. For example:

```
(ssget "X" '((0 . "CIRCLE") (-4 . ">=") (40 . 2.0)))
```

selects all circles whose radius (group code 40) is greater than or equal to 2.0.

The following table shows the possible operators:

Table 4–14. Relational operators for selection set filter lists

Operator	Meaning
"*"	Anything goes (always true)
"="	Equals
"!="	Not equals
"/="	
"<>"	
"<"	Less than
"<="	Less than or equal
">"	Greater than
">="	Greater than or equal
"&"	Bitwise AND (integer groups only)
"&="	Bitwise masked equals (integer groups only)

The use of relational operators depends on the kind of group you are testing:

- All relational operators except for the bitwise operators ("&" and "&=") are valid for both real- and integer-valued groups.
- The bitwise operators "&" and "&=" are valid only for integer-valued groups. The bitwise AND, "&", is true if ((*integer_group* & *filter*) /= 0)—that is, if any of the bits set in the mask are also set in the integer group. The bitwise masked equals, "&=", is true if ((*integer_group* & *filter*) = *filter*)—that is, if all bits set in the mask are also set in the *integer_group* (other bits might be set in the *integer_group* but are not checked).
- For point groups, the *X*, *Y*, and *Z* tests can be combined into a single string, with each operator separated by commas (e.g., ">,>,*"). If an operator is omitted from the string (for example, "=,<>" leaves out the *Z* test), the "anything goes" operator, "*", is assumed.
- Direction vectors (group type 210) can be compared only with the operators "*", "=", and "!=" (or one of the equivalent "not equals" strings).

- You can't use the relational operators with string groups; use wild card tests instead.

Logical Grouping of Filter Tests

The relational operators described in the previous subsection are binary operators. You can also test groups by creating nested Boolean expressions that use the grouping operators shown in the following table. The grouping operators are specified by –4 groups, like the relational operators. They are paired, and must be balanced correctly in the filter list or the **ssget** call fails. The number of operands these operators can enclose depends on the operation, as shown in the table.

Table 4–15. Grouping operators for selection set filter lists

Starting operator	Encloses	Ending operator
`"<AND"`	One or more operands	`"AND>"`
`"<OR"`	One or more operands	`"OR>"`
`"<XOR"`	Two operands	`"XOR>"`
`"<NOT"`	One operand	`"NOT>"`

With the grouping operators, an *operand* is a entity-field group, a relational operator followed by an entity-field group, or a nested expression created by these operators.

An example of grouping operators in a filter list follows:

```
(ssget "X" '((-4 . "<OR")
                (-4 . "<AND")
                   (0 . "CIRCLE")
                   (40 . 1.0)
                (-4 . "AND>")
                (-4 . "<AND")
                   (0 . "LINE")
                   (8 . "ABC")
                (-4 . "AND>")
             (-4 . "OR>"))
)
```

This selects all circles with radius 1.0, plus all lines on layer `"ABC"`.

The grouping operators aren't case-sensitive: you can also use their lowercase equivalents: `"<and"`, `"and>"`, `"<or"`, `"or>"`, `"<xor"`, `"xor>"`, `"<not"`, and `"not>"`.

(sslength *ss*)

This function returns an integer containing the number of entities in selection set *ss*. The number is returned as a real if it is greater than 32,767. Selection sets never contain duplicate selections of the same entity. For example:

```
(setq sset (ssget "L"))    Places the last object in selection set sset
(sslength sset)            returns   1
```

(ssmemb *ename ss*)

This function tests whether entity name *ename* is a member of selection set *ss*. If it is, **ssmemb** returns the entity name (*ename*). If not, it returns `nil`. For example, given that entity name `e1` is a member of selection set `ss1` and entity name `e2` is not, then

```
(ssmemb e1 ss1)          Returns entity name e1
(ssmemb e2 ss1)          Returns nil
```

(ssname *ss index*)

This function returns the entity name of the *index*'th element of selection set *ss*. If *index* is negative or greater than the highest numbered entity in the selection set, `nil` is returned. The first element in the set has an index of zero. Entity names in selection sets obtained with **ssget** will always be names of main entities. Subentities (Block attributes and Polyline vertices) are not returned (but see **entnext**, later, which allows access to them).

For example:

```
(setq sset (ssget))              Creates a selection set named sset
(setq ent1 (ssname sset 0))      Gets name of first entity in sset
(setq ent4 (ssname sset 3))      Gets name of fourth entity in sset
```

To access entities beyond the 32767th one in a selection set, you must supply the *index* argument as a real. For example:

```
(setq entx (ssname sset 50843.0))    Gets name of 50844th entity in sset
```

(strcase *string [which]*)

The **strcase** function takes the string specified by the *string* argument and returns a copy with all alphabetic characters converted to upper- or lowercase, depending on the second argument, *which*. If *which* is omitted or evaluates to `nil`, all alphabetic characters in *string* are converted to uppercase. If *which* is supplied and is not `nil`, all alphabetic characters in *string* are converted to lowercase. For example:

```
(strcase "Sample")       returns    "SAMPLE"
(strcase "Sample" T)     returns    "sample"
```

The **strcase** function will correctly handle case mapping of the currently configured character set (see "Foreign Language Support" on page 184).

(strcat *string1 [string2]* ...)

This function returns a string that is the concatenation of *string1*, *string2*, and so on. For example:

```
(strcat "a" "bout")      returns    "about"
(strcat "a" "b" "c")     returns    "abc"
(strcat "a" "" "c")      returns    "ac"
```

(strlen *[string] ...*)

This function returns an integer that is the number of characters, in `string` . If multiple `string` arguments are provided, it returns the sum of the lengths of all arguments. Omitting the arguments or entering an empty string (as shown in the last two examples) returns the integer 0 (zero).

```
(strlen "abcd")                  returns   4
(strlen "ab")                    returns   2
(strlen "one" "two" "three")     returns   11
(strlen)                         returns   0
(strlen "")                      returns   0
```

(subst *newitem olditem list*)

This function searches `list` for `olditem`, and returns a copy of `list` with `newitem` substituted in place of *every* occurrence of `olditem`. If `olditem` is not found in `list`, **subst** returns `list` unchanged. For example, given

```
(setq sample '(a b (c d) b))
```

then

```
(subst 'qq 'b sample)                returns   (A QQ (C D) QQ)
(subst 'qq 'z sample)                returns   (A B (C D) B)
(subst 'qq '(c d) sample)            returns   (A B QQ B)
(subst '(qq rr) '(c d) sample)       returns   (A B (QQ RR) B)
(subst '(qq rr) 'z sample)           returns   (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. For instance, given

```
(setq who '((first john) (mid q) (last public)))
```

then

```
(setq old
    (assoc 'first who)
)                              returns   (FIRST JOHN)
(setq new '(first j))          returns   (FIRST J)
(subst new old who)            returns   ((FIRST J) (MID Q) (LAST PUBLIC))
```

(substr *string start [length]*)

This function returns a substring of `string`, starting at the `start` character position of `string` and continuing for `length` characters. If `length` is not specified, the substring continues to the end of `string`. The `string` argument (and `length`, if present) must be a positive integer.

You should note that the first character of *string* is character number 1. This differs from all other functions that deal with elements of a list (like **nth**, **ssname**, and so on) which count the first element as 0.

Example

```
(substr "abcde" 2)      returns    "bcde"
(substr "abcde" 2 1)    returns    "b"
(substr "abcde" 3 2)    returns    "cd"
```

(tablet *code [row1 row2 row3 direction]*)

This function is used to retrieve and establish digitizer calibrations. It can be used simply for saving and restoring calibrations, or for creating new tablet transformations.

Depending on the integer specified by *code*, **tablet** either retrieves the current digitizer (tablet) calibration or sets the calibration. If *code* is 0, **tablet** returns the current calibration. If *code* is 1, it must be followed by the new calibration settings: *row1, row2, row3,* and *direction*.

code An integer.

If the *code* you pass equals 0, **tablet** returns the current calibration; in this case, the remaining arguments must be omitted. If the *code* you pass equals 1, **tablet** sets the calibration according to the arguments that follow; in this case, you *must* provide the other arguments.

row1, row2, row3 Three 3D points. These three arguments specify the three rows of the tablet's transformation matrix.

direction A 3D point. This is the vector (expressed in the World Coordinate System, WCS) that is normal to the plane that represents the surface of the tablet.

Note: If the *direction* specified isn't normalized, **tablet** corrects it, so the *direction* it returns when you set the calibration may differ from the value you passed. In a similar way, the third element in *row3* (*Z*) should always equal 1: **tablet** returns it as 1 even if the *row3* in the list specified a different value.

If **tablet** fails, the function returns `nil` and sets the system variable ERRNO to a value that indicates the reason for the failure (see appendix C). This can happen if the digitizer is not a tablet.

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), it will be seen by AutoCAD as the point in your drawing with those same coordinates.

Note: The system variable TABMODE allows an AutoLISP routine to toggle the tablet On and Off.

See also: "Tablet Calibration" on page 41 for additional information on the tablet transformation matrix.

(tblnext *table-name [rewind]*)

This function is used when scanning an entire symbol table. The first argument is a string identifying the symbol table of interest. Valid `table-name` names are `"LAYER"`, `"LTYPE"`, `"VIEW"`, `"STYLE"`, `"BLOCK"`, `"UCS"`, `"APPID"`, `"DIMSTYLE"`, and `"VPORT"`. The string need not be uppercase.

When **tblnext** is used repeatedly, it normally returns the next entry in the specified table each time. (The **tblsearch** function, described next, can set the *next* entry to be retrieved.) However, if the `rewind` argument is present and evaluates to a non-`nil` value, the symbol table is rewound and the first entry in it is retrieved. If there are no more entries in the table, `nil` is returned. Deleted table entries are never returned.

If an entry is found, it is returned as a list of dotted pairs of DXF-type codes and values, similar to those returned by **entget**. For instance:

```
(tblnext "layer" T)                          Retrieves first layer
```

might return

```
(  (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 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). Thus, given a Block called BOX:

```
(tblnext "block")                             Retrieves Block definition
```

might return

```
(  (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: 40000126>)             First entity
)
```

The entity name in the –2 group is accepted by **entget** and **entnext**, but not by all of the 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.

Note: If a Block contains no entities, the –2 group returned by **tblnext** is the entity name of it's Endblk entity.

(tblsearch *table-name symbol [setnext]*)

This function searches the symbol table identified by `table-name` (same as for **`tblnext`**), looking for the symbol name given by `symbol`. Both names are converted to uppercase automatically. If it finds an entry for the given symbol name, it returns that entry in the format described for **`tblnext`**. If no such entry is found, it returns `nil`. For instance:

```
(tblsearch "style" "standard")        Retrieves text style
```

might return

```
( (0 . "STYLE")                       Symbol type
  (2 . "STANDARD")                    Symbol name
  (70 . 0)                            Flags
  (40 . 0.0)                          Fixed height
  (41 . 1.0)                          Width factor
  (50 . 0.0)                          Obliquing angle
  (71 . 0)                            Generation flags
  (3 . "txt")                         Primary font file
  (4 . "")                            Bigfont file
)
```

Normally, **`tblsearch`** has no effect on the order of entries retrieved by **`tblnext`**. However, if **`tblsearch`** is successful and the `setnext` argument is present and non-`nil`, the **`tblnext`** entry counter is adjusted so that the following **`tblnext`** call returns the entry after the one returned by this **`tblsearch`** call.

See also: "Symbol Table Access" on page 69.

(terpri)

This function prints a newline on screen and returns `nil`. The **`terpri`** function is not used for file I/O. To write a newline to a file, use **`print`** or **`princ`**.

(textbox *elist*)

This function measures a desired text entity and returns the diagonal coordinates of a box that encloses the text.

The `elist` must define a text entity. If fields that define text parameters other than the text itself are omitted from `elist`, the current (or default) settings are used. If **`textbox`** is successful it returns a list of two points; otherwise it returns `nil`.

The minimum list accepted by **`textbox`** is that of the text itself.

`(textbox '((1 . "Hello world.")))` might return ((0.0 0.0 0.0) (0.8 0.2 0.0))

In this case, **`textbox`** would use the current defaults for text to supply the remaining parameters.

The points returned by **`textbox`** describe the bounding box of the Text entity 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 entity is oblique, vertical, or 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.

See also: "Text Box Utility Function" on page 26.

(textpage)

On single-screen AutoCAD installations, this function clears the AutoCAD text window and displays it in front of the graphics window. The **textpage** function is equivalent to **textscr**, except it clears any text that was previously displayed in the text window. This function always returns `nil`.

Related topics: See **textscr** described next and **graphscr** on page 123.

(textscr)

On single-screen systems the **textscr** function switches from the graphics screen to the text screen (like the AutoCAD Flip Screen function key). The **textscr** function always returns `nil`.

Related topics: See **textpage** described previously and **graphscr** on page 123.

(trace *function ...*)

This function is a debugging aid. It 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. For example:

```
(trace my-func)   returns   MY-FUNC
```

and sets the trace flag for function **MY-FUNC**. The **trace** function returns the last function name passed to it.

Related topics: See **untrace** on page 165.

(trans *pt from to [disp]*)

This function translates a point (or a displacement) from one coordinate system to another. The *pt* argument is a list of three reals, which can be interpreted as either a 3D point or a 3D displacement (vector). The *from* argument indicates the coordinate system in which *pt* is expressed, and *to* specifies the desired coordinate system for the returned point. The optional *disp* argument, if present and non-`nil`, specifies that *pt* is to be treated as a 3D displacement rather than as a point. The *from* and *to* arguments can be any of the following:

- An integer code from the following table.

Table 4–16. Coordinate system codes

Code	Coordinate system
0	World (WCS)
1	User (current UCS)
2	Display: DCS of current viewport when used with code 0 or 1 DCS of current model space viewport when used with code 3
3	Paper space DCS (used *only* with code 2)

- An entity name, as returned by the **entnext**, **entlast**, **entsel**, **nentsel**, and **ssname** functions. This lets you translate a point to and from the Entity Coordinate System (ECS) of a particular entity. (For some entities, the ECS is equivalent to the WCS; for these entities, conversion between ECS and WCS is a null operation.)
- A 3D extrusion vector (a list of three reals). This is another method of converting to and from an entity's ECS. However, this does not work for those entities whose ECS is equivalent to the WCS.

  The **trans** function returns a 3D point (or displacement) in the requested *to* coordinate system. For example, given a UCS that is rotated 90 degrees counterclockwise around the World *Z* axis

```
(trans '(1.0 2.0 3.0) 0 1)      returns   (2.0 -1.0 3.0)
(trans '(1.0 2.0 3.0) 1 0)      returns   (-2.0 1.0 3.0)
```

  The coordinate systems are discussed in greater detail in "Coordinate System Transformations" on page 36.

For example, if you wanted to draw a line from the insertion point of a piece of text (without using Osnap), you'd convert the Text entity's insertion point from the Text entity's ECS to the UCS

```
(trans text-insert-point text-ename 1)
```

and feed the result to the From point: prompt.

Conversely, you must convert point (or displacement) values to their destination ECS before feeding them to **entmod**. For example, if you wanted to move a Circle (without using the MOVE command) by the UCS-relative offset (1,2,3), you'd need to convert the displacement from the UCS to the Circle's ECS:

```
(trans '(1 2 3) 1 circle-ename)
```

Then you'd add the resulting displacement to the Circle's centre 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'd convert the user's point from the UCS to the DCS

```
(trans user-point 1 2)
```

and each of the Line endpoints from the Line's ECS 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 *filling in* the *Z* coordinate with an appropriate value. The *Z* component used depends on the *from* coordinate system that was specified and whether the value is to be converted as a point or 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* is determined as follows:

Table 4–17. Converted 2D point Z values

From	Filled in Z value
WCS	0.0
UCS	current elevation
ECS	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 *item*)

This function returns the type of *item*, where the type is one of the following items (as an atom). Items that evaluate to `nil` (such as an unassigned symbol) return `nil`.

Table 4–18. Symbol types

Type	Meaning	Type	Meaning
REAL	Floating point numbers	SUBR	Internal functions
FILE	File descriptors	EXSUBR	External functions (ADS)
STR	Strings	PICKSET	Selection sets
INT	Integers	ENAME	Entity names
SYM	Symbols	PAGETB	Function paging table
LIST	Lists (and user functions)		

For example, given the 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 example illustrates how you might use the **type** function.

```
(defun isint (a)
   (if (= (type a) 'INT)          is TYPE integer?
      T                           yes, return T
      nil                         no, return nil
   )
)
```

(untrace *function* ...)

This function clears the trace flag for the specified *functions*, and returns the last function name. It selectively disables the trace debugging aid. For example, the following code clears the trace flag for function MY-FUNC:

```
(untrace my-func)       returns     MY-FUNC
```

Related topics: See **trace** on page 162.

(ver)

This function returns a string that contains the current AutoLISP version number. It should be used (with **equal**) to check compatibility of programs. The string takes this form:

```
"AutoLISP Release X.X"
```

where *X.X* is the current version number. For example:

```
(ver)       might return      "AutoLISP Release 12.0"
```

Applications can tell what version of AutoLISP is being used by examining the string returned by **ver**.

(vmon)

This function is no longer needed to allow virtual function paging, but remains valid to provide compatibility with previous versions. See "Virtual Function Paging" on page 180.

(vports)

This function returns a list of viewport descriptors for the current viewport configuration. Each viewport descriptor is a list consisting of the viewport identification number and the coordinates of the viewport's lower-left and upper-right corners.

If the AutoCAD system variable TILEMODE is set to 1 (on), the returned list describes the viewport configuration created with the AutoCAD VIEWPORTS 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 entities created with the MVIEW command. The viewport entity corners are expressed in paper space coordinates. Viewport number 1 is always paper space when TILEMODE is off.

For example, given a single-viewport configuration with TILEMODE on, the **vports** function might return this:

```
((1 (0.0 0.0) (1.0 1.0)))
```

Similarly, given four equal-sized viewports located in the four corners of the screen and TILEMODE on, the **vports** function might return this:

```
(    (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 *string pattern*)

This function performs a wild card pattern match on `string`. The `string` is compared to the `pattern` to see if they match. If so, `T` is returned; otherwise, `nil` is returned.

Both `string` and `pattern` can be either a quoted string or a variable. The `pattern` can contain the following wild card pattern matching characters. Only the first 500 characters (approximately) of the `string` and `pattern` are compared; anything beyond that is ignored.

Table 4–19. 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. It can be used anywhere in the search pattern: at the beginning, middle, or end
? (Question mark)	Matches any single character

Table 4–19. Wild card characters (continued)

Character	Definition
~ (Tilde)	If it is the first character in the pattern, then 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)

For example:

```
(wcmatch "Name" "N*")                returns      T
```

This tests the string `Name` to see if it begins with the character `N`. You can use commas in a pattern to enter more than one pattern condition. This example performs three comparisons:

```
(wcmatch "Name" "???,~*m*,N*")       returns      T
```

If any of the three pattern conditions is met, **wcmatch** returns `T`. In this case the tests are these: `Name` has three characters (false); Name does not contain an `m` (false); and `Name` begins with `N` (true). At least one condition was met, so this expression returns `T`.

The compare is case sensitive, so upper- and lowercase characters must match. It is valid to use variables and values returned from AutoLISP functions for *`string`* and *`pattern`* values.

If you need 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 this:

```
(wcmatch "Name" "*`,*")                returns      nil
```

Caution: Because other wild card characters might be added in future releases of AutoLISP, it is a good idea to escape all nonalphanumeric characters in the pattern to ensure upward compatibility.

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`, you enter this:

```
(wcmatch "Name" "*`\\*")               returns      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 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 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 only works 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 follows a leading tilde (as in "[ ]ABC]" or "[~]ABC]").

(while *testexpr expr ...*)

This function evaluates *testexpr* and if *testexpr* is not nil, evaluates the other *exprs* and then evaluates *testexpr* again. This continues until *testexpr* is nil. The **while** function then returns the most recent value of the last *expr*. For example, given

```
(setq test 1)
```

then

```
(while (<= test 10)
    (some-func test)
    (setq test (1+ test))
)
```

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.

(write-char *num [file-desc]*)

This function writes one character to the screen or to the open file described by *file-desc*. The *num* argument is the decimal ASCII code for the character to be written and is also the value returned by **write-char**. For example:

```
(write-char 67)        returns     67
```

and writes the letter C on the screen. Assuming that f is the descriptor for an open file:

```
(write-char 67 f)      returns     67
```

and writes the letter C to that file.

The various operating systems under which AutoCAD and AutoLISP run use several different conventions to signal the end of a line in an ASCII text file. UNIX systems, for example, use a single newline character (LF, ASCII code 10), whereas DOS systems use a pair of characters (CR/LF, ASCII codes 13 and 10) for the same purpose. To facilitate development of AutoLISP programs that work in a portable fashion on all supported operating systems, **write-char** translates a newline character (ASCII code 10) into the end-of-line character (or character sequence) used by the operating system you're currently using. Thus, on a DOS system:

```
(write-char 10 f)      returns     10
```

but writes the character sequence CR/LF (ASCII codes 13 and 10) to the file. **write-char** cannot write a NUL character (ASCII code 0) to a file.

See also: Appendix F for a list of ASCII codes.

(write-line *string [file-desc]*)

This function writes `string` to the screen or to the open file described by `file-desc`. It returns `string` quoted in the normal manner, but omits the quotes when writing to the file. For example, assuming that `f` is a valid open file descriptor:

```
(write-line "Test" f)  writes  Test  and returns  "Test"
```

(xdroom *ename*)

This function returns the amount of extended entity data space that is available for the entity *ename*. If unsuccessful, **xdroom** returns `nil`.

Because there is a limit (currently, 16 kilobytes) on the amount of extended data that can be assigned to an entity, and because multiple applications can append extended data to the same entity, this function is provided so an application can verify that there is room for the extended data it wishes to append. It can be called in conjunction with **xdsize**, which returns the size of an extended data list.

Here is an example that looks up the available space for extended entity data of a Viewport entity. Assuming the variable `vpname` contains the name of a Viewport entity,

```
(xdroom vpname)  returns  16162
```

In this example, 16,162 bytes of the original 16,383 bytes of extended entity data space are available, meaning 221 bytes are used. The amount of extended data space that is used can be looked up directly using the **xdsize** function.

(xdsize *list*)

This function returns the size (in bytes) that `list` occupies when it is appended to an entity as extended entity data . If unsuccessful, this function returns `nil`.

The `list` argument must be a valid list of extended entity data that contain an application name previously registered using the **regapp** function. Brace fields (group code 1002) must be balanced. An invalid `list` 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 CMDECHO is on):

```
Invalid application name in 1001 group
```

The `list` can start with a -3 group code (the extended data sentinel), but it is not required. Because extended entity data can contain information from multiple applications, the list must have a set of enclosing parentheses.

For example:

```
(-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 . "}")
    )
)
```

This is an *invalid* **xdsize** list because there are no enclosing parentheses:

```
("MYAPP" (1000 . "SUITOFARMOR")   WRONG
         (1002 . "{")
         (1040 . 0.0)
         (1040 . 1.0)
         (1002 . "}")
)
```

Here is an example in which **xdsize** is sent a list that contains extended entity data from two registered applications:

```
(setq n1 (list "MYAPP"   (cons 1000 "SUITOFARMOR")
                         (cons 1040 0.0)
                         (cons 1040 1.0)
         )
)
(setq n2 (list "YOURAPP" (cons 1000 "SUITOFARMOR")
                         (cons 1040 0.0)
                         (cons 1040 1.0)
         )
)
(regapp "MYAPP")
(regapp "YOURAPP")
(xdsize (list n1 n2))           returns       48
```

(xload *application [onfailure]*)

This function loads an AutoCAD Development System (ADS) application. If the application is successfully loaded, the application name is returned; otherwise, an error message is issued. This function fails if you attempt to load an application that is already loaded.

The `application` argument is entered as a quoted string or a variable that contains the name of an executable file. At the time the file is loaded, it is verified to be a valid ADS application. Additionally, the version of the ADS program, ADS itself, and the version of AutoLISP running are checked for compatibility.

`(xload "/myapps/ame")`	if successful, returns	`"/myapps/ame"`

If the xload operation fails, it normally causes an AutoLISP error. However, if the `onfailure` argument is supplied, **`xload`** returns the value of this argument upon failure instead of issuing an error message. This feature of **`xload`** is similar to that in the **`load`** function.

Related topics: See the Introduction to the *AutoCAD Development System Programmer's Reference* for more information.

(xunload *application [onfailure]*)

This function unloads an ADS application. If the application is successfully unloaded, the application name is returned; otherwise, an error message is issued.

Enter `application` as a quoted string or a variable containing the name of an application that was loaded with the **`xload`** function. The application name must be entered exactly as it was entered for the **`xload`** function. If a path (directory name) was entered for the application in **`xload`**, it can be omitted in the **`xunload`** function.

For example, the following function will successfully unload the application loaded by the **`xload`** function shown previously.

`(xunload "ame")`	if successful, returns	`"ame"`

If the **`xunload`** operation fails, it normally causes an AutoLISP error. However, if the `onfailure` argument is supplied, **`xunload`** returns the value of this argument upon failure instead of issuing an error message. This feature of **`xunload`** is similar to that in the **`load`** function.

Related topics: See the Introduction to the *AutoCAD Development System Programmer's Reference* for more information.

(zerop *item*)

This function returns `T` if `item` is a real or integer and evaluates to zero; otherwise it returns `nil`. It is not defined for other `item` types. For example:

`(zerop 0)`	returns	`T`
`(zerop 0.0)`	returns	`T`
`(zerop 0.0001)`	returns	`nil`

ADS Defined AutoLISP Functions

The following functions are defined by the ADS program *acadapp* (this has the extension *.exp* on DOS platforms) and are only available if that program is loaded. Before calling these functions, you might want to use the **xload** function to verify that *acadapp* is available.

(acad_colordlg *colornum [flag]*)

Displays the standard AutoCAD colour selection dialogue box.

The *colornum* argument is an integer in the range 0–256. It specifies the AutoCAD colour number to display as the initial default. If the optional *flag* argument is supplied and nil the BYLAYER and BYBLOCK buttons are disabled; if it is not supplied or is non-nil these buttons are enabled.

The **acad_colordlg** function returns the colour number that the user selects via the OK button. If the user cancels the dialogue box, **acad_colordlg** returns nil.

Example

The following code prompts the user to select a colour. It specifies a default of green:

```
(acad_colordlg 3)
```

Note: A *colornum* value of 0 defaults to BYBLOCK and a value of 256 defaults to BYLAYER.

(acad_helpdlg *helpfile topic*)

Displays the standard AutoCAD Help dialogue box using a specified file. You can call this function from your AutoLISP routine to provide help on a standard AutoCAD command or your own application specific help.

The *helpfile* argument is a string that specifies an AutoCAD help file (the *.hlp* filename extension is optional). The *topic* argument is a keyword that specifies the topic the dialogue box initially displays. If the *topic* argument is an empty string (""), the Help dialogue box displays the introductory part of the help file.

For your own applications, you will usually specify a customized help file. Help file format is described in chapter 2 of the *AutoCAD Customization Manual*.

Examples

You can create the following file, called *achelp.hlp,* which is an AutoCAD help file (AutoCAD help files must have a *.hlp* extension):

```
The acad_helpdlg function displays the standard AutoCAD
Help dialogue box. The calling format is:

    acad_helpdlg <helpfile> <topic>
\HELPFILE
The <helpfile> argument specifies an AutoCAD help file.
The .hlp filename extension is optional.
\TOPIC
The <topic> argument specifies the topic to initially
display.

If the <topic> argument is empty ("") acad_helpdlg displays
the introductory part of the specified help file.
```

The following code calls **acad_helpdlg** to display the introductory text in the help file *achelp.hlp*:

```
(acad_helpdlg "achelp" "")
```

The following code is almost the same, but displays the help page associated with the TOPIC keyword:

```
(acad_helpdlg "achelp" "topic")
```

(acad_strlsort *list*)

Sorts a list of strings by alphabetical order. The `list` argument is the list of strings to be sorted. The **acad_strlsort** function returns a list of the same strings in alphabetical order.

If the argument list isn't well formed or if there isn't enough memory to do the sort, **acad_strlsort** returns `nil`.

Example

The following code sorts the list of abbreviated month names:

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

and returns the following list:

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

ADS Defined Commands

This section lists the ADS defined AutoCAD commands that have a special means of access from AutoLISP. They are available only when the *acadapp* ADS program is loaded.

(c:bhatch *pt [ss] [vector]*)

Uses the BHATCH command to hatch a selected area.

The first argument to this function is a point `pt` that is an internal point of the area to receive a boundary hatch; this point (if valid) produces a Polyline boundary that defines the hatching area. The `ss` argument is a selection set that provides additional boundary entities. The last argument `vector` is a point list that describes the direction vector BHATCH uses for ray casting, if this argument is not provided it defaults to (0 0), the "Nearest" method. The `vector` argument accepts a 2D or 3D point list; however, if a 3D point is supplied, the *Z* coordinate is ignored.

The *AutoCAD Reference Manual* describes the BHATCH command and the concept of ray casting in greater detail.

Examples

The following examples assume that a valid Hatch pattern name has been set to the HPNAME system variable.

`(setq p1 '(3 5)`
`      ss1 (entlast))`
	
	Selects the last entity created

`(c:bhatch p1)`	*Hatches the area defined by the boundary Polyline created by the point (3,5)*
`(c:bhatch p1 ss)`	*Hatches the area defined by both the boundary Polyline created by the point (3,5) and the selection set* `ss`
`(c:bhatch p1 '(1 0))`	*Hatches the area defined by the boundary Polyline created by the point (3,5) using ray casting of +X*

The coordinate values used in the `vector` argument are real numbers and can be of any value. The following table provides examples of values that match those used in the BHATCH dialogue box.

Table 4–20. Ray casting direction and vector values

Vector value	Direction
(0 0)	(Nearest)
(1 0)	+X (0 deg.)
(0 1)	+Y (90 deg.)
(–1 0)	–X (180 deg.)
(0 –1)	–Y (270 deg.)
(1000 1732)	60 deg. (approx.)
(1 –1)	315 deg.

If successful, the **c:bhatch** function returns the entity name of the Hatch created, and nil if it fails. If **c:bhatch** fails, an error message is available by calling the **bherrs** function.

(c:bpoly *pt [ss] [vector]*)

Uses the BPOLY command to create a boundary Polyline.

The first argument to this function is a point *pt* that is an internal point of the area to receive the boundary Polyline. The *ss* argument is a selection set that provides additional boundary entities. The last argument *vector* is a point list that describes the direction vector BPOLY uses for ray casting (see table 4–20); if this argument is not provided, it defaults to (0 0), the "Nearest" method. The *vector* argument accepts a 2D or 3D point list; however, if a 3D point is supplied, the *Z* coordinate is ignored.

The *AutoCAD Reference Manual* describes the BPOLY command and the concept of ray casting in greater detail.

If successful, the **c:bpoly** function returns the entity name of the boundary Polyline created, and nil if it fails. If **c:bpoly** fails, an error message is available by calling the **bherrs** function.

(bherrs)

Gets an error message generated by a failed call to **c:bhatch** or **c:bpoly**. If successful, it sets the result to contain the message string; otherwise, the result is nil.

Example

After invoking **c:bhatch**, your program can include the following error-checking code:

```
(if (bherrs) (princ (car (bherrs))))
```

If the call failed because HPNAME wasn't initialized, **bherrs** returns the string "bhatch: no hatch pattern defined\n", which the **princ** call displays at the AutoCAD prompt line.

(c:psdrag *mode*)

Invokes the PSDRAG command to set the PSDRAG value. The *mode* argument is an integer that should equal either 0 or 1. The current value of PSDRAG affects interactive use of the PSIN command. If PSDRAG is 1, PSIN generates the PostScript image as the user drags it to scale it. If PSDRAG is 0, PSIN generates and drags only the bounding box of the image. If successful, the **c:psdrag** function returns the new value of PSDRAG, and nil if it fails.

See chapter 14 of the *AutoCAD Reference Manual* and chapter 10 of the *AutoCAD Customization Manual* for more information.

Example

The following code turns on PSDRAG by setting it to 1. The next interactive invocation of PSIN generates the PostScript image as the user drags it during scaling.

```
(c:psdrag 1)
```

(c:psfill *ent pattern [arg1 [arg2]] ...*)

Invokes the PSFILL

The `ent` argument is the name of the Polyline. The `pattern` argument is a string containing the name of the fill pattern. The `pattern` string must be identical to the name of a fill pattern defined in the current *acad.psf* file. The `arguments` are arguments to the internal PostScript fill procedure: their number and type correspond to the arguments required by `pattern`, as defined in *acad.psf*. Each argument is either an integer or a real value. There can be from 0 to 25 arguments per pattern. If the call specifies fewer arguments than the pattern defines, the pattern's default values are used for the remaining arguments. If successful, **`c:psfill`** returns `T`; it returns `nil` if it fails.

Example

The Greyscale fill pattern has a single argument. The following call uses the default Greyscale argument, 50 percent:

```
(c:psfill ename "Greyscale")
```

This call specifies a 10 percent greyscale instead:

```
(c:psfill ename "Greyscale" 10)
```

See: Chapter 10 of the *AutoCAD Customization Manual* for more information.

(c:psin *filename position scale*)

Invokes the PSIN command to import an encapsulated PostScript (*.eps*) file. The `filename` argument is a string that contains the name of the PostScript image (you don't have to specify the *.eps* filename extension). The `position` argument is a point specifying the insertion point of the (anonymous) PostScript block. The `scale` argument is a real value specifying the scale factor. If successful, **`c:psin`** returns the name of the newly created entity; it returns `nil` if it fails.

Example

The following code imports a PostScript file called *sample.eps,* inserts it at (24,19), and scales it with a factor of 25:

```
(c:psin "sample" '(24 19) 25)
```

Chapter 5
Memory Management and Programming Techniques

Memory Management

AutoCAD is designed to adjust its memory usage as necessary. Although some systems with limited memory may require fine-tuning, most users will never need the information provided in this section.

All symbols, user-defined functions, and the standard functions this guide describes are stored in your computer's memory for the duration of the AutoCAD editing session only. When AutoLISP starts up, it acquires two large areas of memory for itself. The first, called the *heap*, is the area where all functions and symbols (also called *nodes*) are stored; the more symbols and functions you have (and the more complex your functions), the more heap space is used. The second area, called the *stack*, holds function arguments and partial results; the deeper you *nest* functions, or the more recursion your functions perform, the more stack space is used.

Node Space

A *node* is a memory structure that can represent all AutoLISP data types. Currently AutoLISP uses 12-byte nodes. To avoid memory fragmentation and heap management overhead, nodes are allocated from the heap in groups called *segments*. By default, a segment is 514 nodes (6168 bytes).

AutoLISP maintains a list of *free* nodes (nodes that are not currently bound to a symbol). When it needs a node in which to store a symbol or value, AutoLISP searches the *free* list to find an available node. If there are none, an automatic garbage collection is performed, placing any nodes no longer bound to a symbol on the *free* list. Then one is chosen to satisfy the request.

If the garbage collection results in too few free nodes, AutoLISP requests one additional segment from the heap. If the request is successful, the new nodes are placed on the *free* list and one of them is chosen to satisfy the original request for a node. If no additional segments can be obtained from the heap, virtual function paging is invoked to free some nodes by swapping out the least recently used function; otherwise, an *insufficient node space* message is displayed and the function requesting node space is aborted. You should note that node space is *never* returned to the heap until you exit AutoCAD.

It is possible to force a garbage collection using the **gc** function:

```
(gc)
```

However, note that garbage collections are time-consuming operations that should not be performed unnecessarily. They are best left to AutoLISP's automatic mechanism, which performs them only when necessary.

Recovering Node Space

You might find you are creating functions and symbols that you need for only a short while. When they're no longer useful, *undefine* them by assigning *nil* to them. For instance, if you've loaded and used a function named **setup** and have no further need of it, you can use

```
(setq setup nil)
```

to get rid of it. Assigning *nil* recovers the node space used by the function, and other functions and symbols can now use it.

If you want to free up the node space used by a symbol of type FILE (as returned by the **open** function), you must **close** it before setting it to nil. There is a finite number of *open file slots*, and failure to close a file causes its slot to remain in use, limiting the number of new files that can be opened.

Note: Prior to Release 12, AutoCAD maintained a list of all defined functions and symbols in the symbol, `atomlist`. This list is no longer maintained in this manner, therefore the method of "chopping the atomlist" is no longer a valid method for clearing out functions and symbols. You can still retrieve a list of defined functions and symbols by using the **atoms-family** function:

```
(atoms-family 0)
```

This function can be used to retrieve the complete list or just selected symbols, see page 95 for more information on the **atoms-family** function.

Technical Notes

Note: *The following information is for the benefit of experienced LISP hackers only. Novices can (and should) ignore this discussion. This section describes internal AutoLISP mechanisms that are subject to change without notice.*

Memory Statistics

The **mem** function

```
(mem)
```

displays the current state of AutoLISP's memory, and returns nil. *Nodes:* is the total number of nodes allocated so far, which should equal the node segment size multiplied by the number of segments. *Free nodes:* is the number of nodes currently on the *free* list placed there by a garbage collection. *Segments:* is the number of node segments allocated, and *Allocate:* is the current segment size. *Collections:* is a count of garbage collections, whether automatic or forced.

String Space

String storage space comes out of the same heap as node segments. If your program calls the manual allocation functions (described later) to allocate all of available memory as nodes, you're *very* likely to get an *insufficient string space* message, which is just as devastating as *insufficient node space.* We recommend you let the automatic allocation of nodes handle this so that you end up with sufficient string space. String space is used for everything from symbol names to prompt strings and menu strings that are passed to AutoLISP for evaluation. If you use long menu items with AutoLISP expressions for evaluation, these need a significant amount of contiguous string space from the heap.

Symbol Storage

The memory structure in AutoLISP is pointer intensive. The use of nodes is pervasive and everything is represented in the node structure. The simple act of setting a symbol equal to a value, as in

```
(setq longsym 3.1415)
```

requires two nodes; one to store the symbol name, and one to store its value. If the symbol name is six or fewer characters long, the name is stored directly in the symbol name node; otherwise, string space is allocated from the heap for storage of the name, and the symbol name node then points to this string. In short, using short symbol names (six or fewer characters) reduces string space requirements and heap fragmentation.

Manual Allocation

You can use the `alloc` and `expand` functions to manually control node and string space allocation and so increase the efficiency of your applications. By using these expressions at the beginning of your *acad.lsp* file, you can preallocate the nodes and also reserve some string space. This can reduce the number of garbage collections, improving the run-time efficiency of your application.

You can use the `alloc` function to alter the size of future segment requests to be something other than 514 nodes apiece.

```
(alloc number)
```

The `alloc` function sets the segment size to *number* nodes and returns the previous setting.

Using the `expand` function, you can manually allocate node space by requesting a specified number of segments.

```
(expand number)
```

number is the number of segments you want to allocate.

The `expand` function returns the number of segments it was able to acquire from the heap, which might be far fewer than requested due to the amount of remaining heap space.

Example:

```
(alloc 1024)
```

sets the segment size to 1024 nodes, which requires 12,488 bytes of heap space for each segment.

```
(alloc 100)
```

sets the segment size to 100, which requires only 1200 bytes per segment.

With the segment size set to the default of 512 nodes, a call to

```
(expand 10)
```

requests 10 segments (61480 bytes). It doesn't hurt to ask for more segments than are available.

Here's a practical example:

```
(alloc 3000)    Sets node segment size to 3000 nodes
(expand 1)      Gets 3000 free nodes (one segment)
(alloc 10000)   Sets large segment size to avoid adding more segments
```

This scheme uses 36,000 bytes for node space, leaving the remainder of the heap for string space. The nodes are preallocated and placed on the *free* list. This means that no garbage collections are necessary until you use up all 3000 nodes. Once you use up these nodes, of course, garbage collections are invoked to satisfy further node requests. The `(alloc 10000)` sets the segment size to a value that prevents additional segments from being allocated from the heap, thereby *reserving* this space for string use.

If you apply the opposite strategy by sequentially reducing the segment size until a request for one node was unsuccessful (i.e., "`(alloc 1) (expand 1)`" returns 0), you use up *all* of your heap for node space, leading to an *insufficient string space* error. This is *not* recommended, since AutoLISP will become useless without available string space.

Virtual Function Paging

Virtual function paging can only take effect after all other types of virtual memory have been exhausted; this seldom happens on most platforms. The **`vmon`** *function is provided for compatibility with previous versions of AutoLISP.*

If your AutoLISP application grows too large to fit in the available node space, you can enable the AutoLISP *virtual function pager* to allow your program to continue to grow. To do this, execute the **`vmon`** function before the first **`defun`** in your program:

```
(vmon)
```

This enables virtual function paging for the remainder of the AutoCAD drawing session. Once you enable it, function paging cannot be turned off. Only functions created via **`defun`** subsequent to the **`vmon`** call are eligible for paging, so if you **`defun`** functions before calling **`vmon`**, they won't be paged out and might still cause *insufficient node space* terminations.

After the **`vmon`** function is executed, AutoLISP pages out infrequently used functions whenever it runs out of node space, and automatically reads them back in when needed. Don't worry about this paging as it is handled automatically and is completely transparent to your program. The functions are swapped to a temporary file which is managed through the AutoCAD file pager.

The virtual memory system only pages *functions;* you must still have sufficient node space to accommodate all data lists, function names, and variable names used by your program.

The **mem** function displays two additional fields when **vmon** is activated. *Swap-ins:* is the number of functions swapped *in* from the page file created for the virtual memory system. These are demand paged functions brought in from the page file when requested in a simple function call. The *Page file:* entry is the size of the page file created to hold the functions that have been paged out.

After the **vmon** function is executed, all **defuns** place a new node called a *page table* at the start of every function list. This node is added before the list containing the formal arguments. Page table nodes are exclusively for the use of the pager, and should not be manipulated in any way by user programs. The **type** function returns PAGETB for these nodes.

When AutoLISP runs out of nodes, the least frequently used function is swapped out by writing it to the paging file, saving the page file address in the page table, and releasing all of the function's nodes following the page table. The page table is marked to indicate that the function is swapped out. When a swapped out function is evaluated, it is read back in from the paging file (possibly swapping out other functions) prior to execution. Once a function has been written out to the paging file, subsequent swapouts simply release its nodes; there's no need to write out the function, since it's already present in the file.

In AutoLISP, functions created with **defun** are just lists, and can be manipulated like any other list. Programs that do this must be cognizant of the operation of the pager (or never use **vmon**). First of all, functions created with **defun** have a page table node in front, so you should skip it if you're scanning the function. If you create a function yourself as a list (bypassing **defun**), it works fine, but won't be eligible for swapping so you can easily run out of memory if you do this a lot. Conversely, you can lock a function in memory by redefining it without its page table. For instance, to lock a function with the name zorp into memory, you could use:

```
(setq zorp (cdr zorp))
```

to delete the leading page table. Page tables print as a space when you **print** the function. You can tell if a function is swappable by checking whether there's a space after the first left parenthesis: if so, it's swappable.

If you're attempting to scan a function as data and it's swapped out, you find only the page table in the function list. Accessing the function doesn't swap it in—only *evaluating* it does this. So if you're constructing functions and modifying them on the fly, build them as lists instead of using **defun**, or use the trick above to lock them into memory.

Good Programming Techniques

This section provides some useful programming techniques for both novices and experienced AutoLISP programmers. It is not intended to be a tutorial; recommended texts on the LISP programming language are mentioned in the introduction on page 1.

General Code Organization

The organizational methods you develop for commenting and indenting your code are completely up to you; these are purely for the benefit of the programmer and the end user and have no bearing on the code itself. If appropriate, you can develop standards for yourself or your group so that the code you develop maintains some consistency and looks professional.

Comments

Using comments in program code is very important. Comments are useful to both the programmer and future users who may need to revise a program to suit their needs. A comment in AutoLISP is anything on a line that follows a semicolon (;). Some of the uses for comments are the following:

- give a title, authorship, and creation date
- provide instructions on using a routine
- make explanatory notes throughout the body of a routine
- make notes to yourself during debugging
- allow for characters that provide visual aesthetics

Liberal commenting is effort well rewarded when writing AutoLISP programs. Both block comments (those that fill entire lines) and in-line comments (those following a semicolon on a line of executable code) are useful. The LISP fraternity has developed workable conventions for different classes of comments (see, for example, “Common LISP” by Guy L. Steele Jr.), and some of the *.lsp* files provided with AutoCAD (but not all of them!) provide good examples of commenting style. The file *ai_utils.lsp* is extensively commented and contains some interesting and useful AutoLISP routines.

Keep the following points in mind when developing a commenting style:

- The description portion of the header can describe the usage of the file in enough detail so it can be used without resorting to other documentation.
- The number of semicolons preceding a comment and a comment’s placement in the file can help to indicate the contents or importance of the comment.
- Many monitors can display only 80 columns of text; if you have a comment starting in the 81st column, it might not be displayed.

The rules of the language are flexible, and it is much more important that comments be present than they obey any particular layout rules.

Indenting and Alignment

The extensive use of parentheses in AutoLISP can make it difficult to read. The traditional technique for combatting this confusion is indentation. This means laying out the program so that the more deeply nested a line of code is, the farther to the right it starts. The parentheses get so deep that typical indentation standards move only a couple of spaces further right for each level, in contrast to the typical C-language standard of four spaces per level.

Indentation rules are harder to specify in prose than they are to indicate by example. The best approach for developing good code layout is reading and emulating other authors' code that you find particularly pleasing and easy to read. Also, numerous LISP editing systems include so-called "pretty-printers," which are programs that read arbitrarily laid out LISP code and reformat it with suitable indentation.

Programming Tips

We cannot cover every aspect of LISP programming techniques in this section; however, we will discuss some of the more important issues that relate to programming in AutoLISP. If you are having a problem with part of your code, it is often helpful to look at other AutoLISP programs to see how they have handled similar situations.

Error Handling

AutoLISP provides a method for dealing with user (or program) errors. With the ***error*** function you can ensure that desired system variables or a particular AutoCAD state is returned to after an unexpected error occurs. Through this user-definable function you can assess the error condition and return an appropriate message to the user. If this function has not been defined or is `nil`, the standard error handler will, upon error, cease AutoLISP evaluation, print an error message, then display a traceback of the calling function and its callers up to 100 levels deep. It is often beneficial to leave this standard error handler in effect while debugging your program.

Before defining your own ***error*** function, it is usually preferable to save the current contents of ***error*** so that the previous error handler can be restored upon exit. When an error condition exists, AutoCAD calls the currently defined ***error*** function and passes it one argument, which is a text string describing the nature of the error. Typically your ***error*** function should be designed to exit quietly after a Ctrl+C or an **exit** function call. The standard way to accomplish this is to include the following statements in your error handling routine:

```
(if (/= msg "Function cancelled")
  (if (= msg "quit / exit abort")
    (princ)
    (princ (strcat "\nError: " msg))
  )
  (princ)
)
```

This code examines the error message passed to it and ensures that if a real error is detected, the user will be informed of the nature of the error, at least as much as AutoLISP itself knows. If the user cancels the routine while it is running, nothing is returned from this code. Likewise, if an error condition is programmed into your code and the `exit` function is called, nothing is returned. It is presumed that you will have already explained the nature of the error via print statements of some sort. Remember to include a terminating call to `princ` if you don't want a return value printed at the end of an error routine.

If you are using UNDO calls to allow your routine to be undone in a single step, you must provide the balancing Undo calls that would be called if your routine exited normally. In fact, you might want to have AutoCAD UNDO everything or the last part of whatever it is that your routine has created. Programs can use the system variables UNDOCTL and UNDOMARKS to determine how (or if at all) it should call UNDO.

The main caveat about error handling routines is that they are AutoLISP functions just like any other, and can be cancelled by the user. So keep them as short and as fast as possible. This will increase the likelihood that all of it will execute if called.

Input Validation

You should protect your code from unintentional user errors. The AutoLISP user input `getxxx` functions do much of this for you, usually guaranteeing that the item entered by the user is at least syntactically correct. However, its easy and dangerous to forget to check for adherence to other criteria that the program requires, but which the service functions themselves can't check. Input validity checking is tedious, but its omission can seriously affect a program's integrity.

Loading

AutoLISP programs are usually provided in the form of *.lsp* files to be loaded by the `load` function. Besides creating functions for the user to call, `load` provides an opportunity for the programmer to tell the user how to use the program. Since AutoLISP evaluates the expressions in the file, if there is a call to the `princ` function (or `print`, `prin1`, or the like), you can display any desired information at the prompt line. You can also suppress what AutoLISP would normally print by making the very last line of the file "`(princ)`."

The loading of a file provides other opportunities for calling AutoLISP functions. Any code in a *.lsp* file that is not part of a `defun` statement will be executed when that file is loaded. You can use this to set up certain parameters or perform any other initialization procedures in addition to displaying desired textual information as described above.

Foreign Language Support

If you develop AutoLISP programs that might be used with a foreign language version of AutoCAD, the standard AutoCAD commands and keywords will be automatically translated if you precede each command or keyword with the underscore character " _ ". The following example demonstrates this:

```
    (command "_line" pt1 pt2 pt3 "_c")
```

Entity Access Functions

You should be aware that the entity access functions are relatively slow. It is usually best to get the contents of a particular entity (or symbol table entry) once and keep that information stored in memory, rather than repeatedly asking AutoCAD for the same data. Just be sure that the data remains valid; if the user has an opportunity to alter the entity or symbol table entry, you should re-issue the entity access function to ensure validity of the data.

Point Transformations

If you're doing point transformations with the **trans** function and need to make that part of a program run faster, you can construct your own transformation matrix on the AutoLISP side by using **trans** once to transform each of the "basis vectors" (0 0 0), (1 0 0), (0 1 0), and (0 0 1). Of course, writing matrix multiplication functions in LISP isn't fun, so you probably shouldn't bother unless your program is doing a lot of transformations.

Unit Conversion

The first time **cvunit** converts to or from a unit during a drawing editor session, it must look up the string that specifies the unit in *acad.unt*. If your application has many values to convert from one system of units to another, it is more efficient to convert the value 1.0 by a single call to **cvunit**, then use the returned value as a scale factor in subsequent conversions. This works for all units defined in *acad.unt* except temperature scales, which involve an offset as well as a scale factor.

Appendix A
AutoLISP and ADS Functions

The table in this appendix shows the current AutoLISP functions and the corresponding ADS functions. It's meant to make it easy for you to compare the two function sets and the argument lists for corresponding functions.

Table A–1. AutoLISP and ADS functions

AutoLISP function	ADS function
(+ *number number* ...)	
(- *number [number]* ...)	
(* *number [number]* ...)	
(/ *number [number]* ...)	
(= *atom atom* ...)	
(/= *atom atom* ...)	
(< *atom atom* ...)	
(<= *atom atom* ...)	
(> *atom atom* ...)	
(>= *atom atom* ...)	
(~ *number*)	
(*error* *string*)	
(1+ *number*)	
(1- *number*)	
(abs *number*)	
(ads)	ads_loaded()
(alloc *number*)	
	ads_abort(*str*)
(alert *string*)	ads_alert(*str*)
(and *expression* ...)	
(angle *pt1 pt2*)	ads_angle(*pt1, pt2*)
(angtof *string [mode]*)	ads_angtof(*str, unit, v*)

Table A–1. AutoLISP and ADS functions (continued)

AutoLISP function	ADS function
(angtos *angle [mode [precision]]*)	ads_angtos(*v, unit, prec, str*)
(append *expr* ...)	
(apply *function list*)	
(ascii *string*)	
(assoc *item alist*)	
(atan *num1 [num2]*)	
(atof *string*)	
(atoi *string*)	
(atom *item*)	
(atoms-family *format [symlist]*)	
(Boole *func int1 int2* ...)	
(boundp *atom*)	
	ads_buildlist(*rtype, [, argument]* ...)
(car *list*)	
(cdr *list*)	
(caar *list*), (cadr *list*), (cddr *list*), (cadar *list*), etc.	
(chr *integer*)	
(close *file-desc*)	
	ads_cmd(*rbp*)
(command *[args]* ...)	ads_command(*rtype, [, argument]* ...)
(cond (*test1 result1* ...) ...)	
(cons *new-first-element list*)	
(cos *angle*)	
(cvunit *value from to*)	ads_cvunit(*value, oldunit, newunit, result*)
(defun *sym argument-list expr* ...)	ads_defun(*sname, funcno*)
(distance *pt1 pt2*)	ads_distance(*pt1, pt2*)
(distof *string [mode]*)	ads_distof(*str, unit, v*)
	ads_draggen(*ss, pmt, cursor, scnf, p*)
(entdel *ename*)	ads_entdel(*ent*)
(entget *ename [applist]*)	ads_entget(*ent*)
	ads_entgetx(*ent, apps*)
(entlast)	ads_entlast(*result*)

Table A–1. AutoLISP and ADS functions (continued)

AutoLISP function	ADS function
(entmake *[elist]*)	ads_entmake(*ent*)
(entmod *elist*)	ads_entmod(*ent*)
(entnext *[ename]*)	ads_entnext(*ent*, *result*)
(entsel *[prompt]*)	ads_entsel(*str*, *entres*, *ptres*)
(entupd *ename*)	ads_entupd(*ent*)
(eq *expr1 expr2*)	
(equal *expr1 expr2 [fuzz]*)	
(eval *expr*)	
(exit)	ads_exit(*status*)
(exp *number*)	
(expand)	
	ads_fail(*str*)
(expt *base power*)	
(findfile *filename*)	ads_findfile(*fname*, *result*)
(fix *number*)	
(float *number*)	
(foreach *name list expr* ...)	
(gc)	
(gcd *num1 num2*)	
(getangle *[pt] [prompt]*)	ads_getangle(*point*, *prompt*, *result*)
	ads_getargs()
(getcorner *pt [prompt]*)	ads_getcorner(*point*, *prompt*, *result*)
(getdist *[pt] [prompt]*)	ads_getdist(*point*, *prompt*, *result*)
(getenv *variable-name*)	
(getfiled *title filename ext flags*)	ads_getfiled(*title*, *default*, *ext*, *flags*, *result*)
	ads_getfuncode()
	ads_getinput(str)
(getint *[prompt]*)	ads_getint(*prompt*, *result*)
(getkword *[prompt]*)	ads_getkword(*prompt*, *result*)
(getorient *[pt] [prompt]*)	ads_getorient(*point*, *prompt*, *result*)
(getpoint *[pt] [prompt]*)	ads_getpoint(*point*, *prompt*, *result*)
(getreal *[prompt]*)	ads_getreal(*prompt*, *result*)

Table A–1. AutoLISP and ADS functions (continued)

AutoLISP function	ADS function
(getstring *[cr] [prompt]*)	**ads_getstring**(*cronly, prompt, result*)
	ads_getsym(*sname, value*)
(getvar *varname*)	**ads_getvar**(*sym, result*)
(graphscr)	**ads_graphscr**()
(grclear)	**ads_grclear**()
(grdraw *from to color [highlight]*)	**ads_grdraw**(*from, to, color, hl*)
(grread *[track [allkeys [curtype]]]*)	**ads_grread**(*track, type, result*)
(grtext *[box text [highlight]]*)	**ads_grtext**(*box, text, hl*)
(grvecs *vlist [trans]*)	**ads_grvecs**(*vlist, mat*)
(handent *handle*)	**ads_handent**(*handle, entres*)
(if *testexpr thenexpr [elseexpr]*)	
	ads_init(*argc, argv*)
(initget *[bits] [string]*)	**ads_initget**(*val, kwl*)
(inters *pt1 pt2 pt3 pt4 [onseg]*)	**ads_inters**(*from1, to1, from2, to2, teston, result*)
	ads_invoke(*args, result*)
	ads_isalnum(*c*)
	ads_isalpha(*c*)
	ads_iscntrl(*c*)
	ads_isdigit(*c*)
	ads_isgraph(*c*)
	ads_islower(*c*)
	ads_isprint(*c*)
	ads_ispunct(*c*)
	ads_isspace(*c*)
	ads_isupper(*c*)
	ads_isxdigit(*c*)
(itoa *int*)	
(lambda *arguments expr ...*)	
(last *list*)	
(length *list*)	
(list *expr ...*)	
(listp *item*)	

Table A–1. AutoLISP and ADS functions (continued)

AutoLISP function	ADS function
(load *filename [onfailure]*)	
(log *number*)	
(logand *number number* ...)	
(logior *integer* ...)	
(lsh *num1 numbits*)	
(mapcar *function list1 ... listn*)	
(max *number number* ...)	
(mem)	
(member *expr list*)	
	ads_link(*cbc*)
(menucmd *string*)	**ads_menucmd(***str*)
(min *number number* ...)	
(minusp *item*)	
(nentsel *[prompt]*)	**ads_nentsel(***str, entres, ptres, xformres, refstkres*)
(nentselp *[prompt] [pt]*)	**ads_nentselp(***str, entres, ptres, flag, xformres, refstkres*)
	ads_newrb(*v*)
(not *item*)	
(nth *n list*)	
(null *item*)	
(numberp *item*)	
(open *filename mode*)	
(or *expr* ...)	
(osnap *pt mode-string*)	**ads_osnap(***pt, mode, result*)
(polar *pt angle distance*)	**ads_polar(***pt, angle, dist, ptres*)
(prin1 *[expr [file-desc]]*)	**ads_printf(***format, [, argument]* ...)
(princ *[expr [file-desc]]*)	
(print *[expr [file-desc]]*)	
(progn *expr* ...)	
(prompt *msg*)	**ads_prompt(***str*)
	ads_putsym(*sname, value*)
(quit)	
(quote *expr*)	

Table A–1. AutoLISP and ADS functions (continued)

AutoLISP function	ADS function
(read *string*)	
(read-char *[file-desc]*)	
(read-line *[file-desc]*)	
(redraw *[ename [mode]]*)	**ads_redraw**(*ent, mode*)
(regapp *application*)	**ads_regapp**(*appname*)
	ads_regfunc(*fhdl, fcode*)
	ads_relrb(*rb*)
(rem *num1 num2 ...*)	
(repeat *number expr ...*)	
	ads_retint(*ival*)
	ads_retlist(*rbuf*)
	ads_retname(*aname, type*)
	ads_retnil()
	ads_retpoint(*pt*)
	ads_retreal(*ival*)
	ads_retstr(*s*)
	ads_rett()
	ads_retval(*rbuf*)
	ads_retvoid()
(reverse *list*)	
(rtos *number [mode [precision]]*)	**ads_rtos**(*val, unit, prec, str*)
(set *sym expr*)	
(setq *sym1 expr1 [sym2 expr2] ...*)	
(setvar *varname value*)	**ads_setvar**(*sym, val*)
(sin *angle*)	
(sqrt *number*)	
(ssadd *[ename [ss]]*)	**ads_ssadd**(*ename, sname, result*)
(ssdel *ename ss*)	**ads_ssdel**(*ename, ss*)
	ads_ssfree(*sname*)
(ssget *[mode] [pt1 [pt2]] [pt-list] [filter-list]*)	**ads_ssget**(*str, pt1, pt2, entmask, ss*)
(sslength *ss*)	**ads_sslength**(*sname, len*)
(ssmemb *ename ss*)	**ads_ssmemb**(*ename, ss*)

Table A–1. AutoLISP and ADS functions (continued)

AutoLISP function	ADS function
(**ssname** *ss index*)	**ads_ssname**(*ss, i, entres*)
(**strcase** *string [which]*)	
(**strcat** *string1 [string2] ...*)	
(**strlen** *[string] ...*)	
(**subst** *newitem olditem list*)	
(**substr** *string start [length]*)	
(**tablet** *code [col1 col2 col3 direction]*)	**ads_tablet**(*list, result*)
(**tblnext** *table-name [rewind]*)	**ads_tblnext**(*tblname, rewind*)
(**tblsearch** *table-name symbol [setnext]*)	**ads_tblsearch**(*tblname, sym, setnext*)
(**terpri**)	
(**textbox** *elist*)	**ads_textbox**(*ent, p1, p2*)
(**textpage**)	**ads_textpage**()
(**textscr**)	**ads_textscr**()
	ads_tolower(*c*)
	ads_toupper(*c*)
(**trace** *function ...*)	
(**trans** *pt from to [disp]*)	**ads_trans**(*pt, from, to, disp, result*)
(**type** *item*)	
(**untrace** *function ...*)	
	ads_undef(*sname, funcno*)
	ads_usrbrk()
(**ver**)	
(**vports**)	**ads_vports**(*result*)
(**wcmatch** *string pattern*)	**ads_wcmatch**(*string, pattern*)
(**while** *testexpr expr ...*)	
(**write-char** *num [file-desc]*)	
(**write-line** *string [file-desc]*)	
(**xdroom** *ename*)	**ads_xdroom**(*ent, result*)
(**xdsize** *list*)	**ads_xdsize**(*xd, result*)
	ads_xformss(*ssname, genmat*)
(**xload** *application [onfailure]*)	**ads_xload**(*app*)
(**xunload** *application [onfailure]*)	**ads_xunload**(*app*)
(**zerop** *item*)	

Appendix B
DXF Group Codes

This appendix is a quick reference to the DXF group codes, which are described in greater detail in chapter 11 of the *AutoCAD Customization Manual.* The first section lists the group codes in numerical order. The second organizes them by entity.

Important: The group codes encountered by an AutoLISP or ADS application differ slightly from the group codes as they appear in a DXF file. This appendix describes the codes from an application's point of view.

Group Codes in Numerical Order

This table shows negative group codes, which don't actually appear in a DXF file but do appear to programs. It also omits some codes that don't appear to programs.

In the table, "(fixed)" indicates that this group code always has the same purpose. The purpose of group codes that aren't fixed can vary depending on context.

Table B–1. Entity group codes by number

Group code	Value type
–4	Conditional operator (used *only* with `(ssget)` and `ads_ssget()`)
–3	Extended entity data (XDATA) sentinel (fixed)
–2	Entity name reference (fixed)
–1	Entity name (changes each time drawing is opened; never saved); (fixed)
0	Starts an entity. The type of entity is given by the text value that follows this group (fixed)
1	The primary text value for an entity
2	A name: Attribute tag, Block name, and so on
3–4	Other textual or name values
5	Entity handle expressed as a hexadecimal string (fixed)
6	Line type name (fixed)

Table B–1. Entity group codes by number (continued)

Group code	Value type
7	Text style name (fixed)
8	Layer name (fixed)
10	Primary point (start point of a Line or Text entity, centre of a Circle, etc.)
11–18	Other points ***Note:*** These are the only coordinate group codes that an application sees. The *Y* (20–28) and *Z* (30–38) coordinates that appear in a DXF file are passed to an application as part of an AutoLISP point list or an ADS result buffer
39	This entity's thickness if nonzero (fixed)
40–48	Floating-point values (text height, scale factors, etc.)
49	Repeated value—multiple 49 groups may appear in one entity for variable-length tables (such as the dash lengths in the LTYPE table). A *7x* group always appears before the first 49 group to specify the table length
50–58	Angles
62	Colour number (fixed)
66	"Entities follow" flag (fixed)
67	Space (that is, model or paper space)
70–78	Integer values such as repeat counts, flag bits, or modes
210	Extrusion direction (fixed) ***Note:*** As with point coordinates, an application sees only the 210 group. The *Y* (220) and *Z* (230) components of an extrusion vector are passed to an application as part of an AutoLISP point list or an ADS result buffer
999	Comments
1000	An ASCII string (up to 255 bytes long) in XDATA
1001	Registered application name (ASCII string up to 31 bytes long) for XDATA (fixed)
1002	XDATA control string ("{" or "}"); (fixed)
1003	Layer name in XDATA
1004	Chunk of bytes (up to 127 bytes long) in XDATA
1005	Entity handle in XDATA
1010	A point in XDATA
1011	A 3D World space position in XDATA
1012	A 3D World space displacement in XDATA

Table B–1. *Entity group codes by number (continued)*

Group code	Value type
1013	A 3D World space direction in XDATA *Note:* Again, these are the only coordinate group codes that an application sees. The *Y* (1020, 1021, 1022, or 1023) and *Z* (1030, 1031, 1032, or 1033) coordinates that appear in a DXF file are passed to an application as part of an AutoLISP point list or an ADS result buffer
1040	Floating-point value in XDATA
1041	Distance value in XDATA
1042	Scale factor in XDATA
1070	16-bit integer in XDATA
1071	32-bit signed long integer in XDATA

Group Codes by Entity

The following table, B-2, shows group codes that apply to virtually all entities (strictly speaking, handles don't appear in tables and group 210 applies only to planar entities; optional codes are shown in gray). When you refer to table B-3 and table B-4, which list the codes associated with an entity, don't forget that the codes shown here can also be present.

Table B–2. Group codes that apply to all entities

Group code	Meaning	If omitted defaults to . . .
–1	Entity name (changes each time drawing is opened)	*Not* omitted
0	Entity type	*Not* omitted
8	Layer name	*Not* omitted
5	Handle (*always* present for entities and Block definitions)	*Not* omitted
6	Linetype name (present if not BYLAYER). The special name BYBLOCK indicates a floating linetype	BYLAYER
39	Thickness (present if nonzero)	0
62	Colour number (present if not BYLAYER). Zero indicates the BYBLOCK (floating) colour. 256 indicates BYLAYER	BYLAYER
67	Absent or zero indicates entity is in model space. One indicates entity is in paper space	0
Other entity-definition groups appear here		
210	Extrusion direction (present if the entity's extrusion direction is not parallel to the World *Z* axis) This group applies to Line, Point, Circle, Shape, Text, Arc, Trace, Solid, Block Reference (Insert), Polyline, Dimension, Attribute, and Attribute Definition entities	(0,0,1)

Caution: Although these tables show the order of group codes as they usually appear, it's not a good idea to write programs that rely on this order, which can change under certain conditions or in a future AutoCAD release. Code to handle an entity should be driven by a case (switch) or a table, so it can process each group correctly even if the order is unexpected.

Entity Group Codes

The following table, B-3, shows group codes for entities (as they would be saved in the ENTITIES section of a DXF file; optional codes are shown in gray). For the codes that apply to Block definitions and table entries, see "Block and Table Group Codes" on page 203.

Table B–3. Entity group codes by entity

Entity type	Group codes	Meaning
3DFACE	10	First corner
	11	Second corner
	12	Third corner
	13	Fourth corner (if only three corners entered, this equals the third corner)
	70	Invisible edge flags (optional; default: 0): 1 First edge is invisible 2 Second edge is invisible 4 Third edge is invisible 8 Fourth edge is invisible
ATTDEF	10	Text start point
	40	Text height
	1	Default value (string)
	3	Prompt string
	2	Tag string
	70	Attribute flags: 1 Attribute is invisible (does not display) 2 This is a constant Attribute 4 Verification is required on input of this Attribute 8 Attribute is preset (no prompt during insertion)
	73	Field length (optional; default: 0)
	50	Text rotation (optional; default: 0)
	41	Relative *X* scale factor (optional; default: 1)
	51	Oblique angle (optional; default: 0)
	7	Text style name (optional; default: STANDARD)
	71	Text-generation flags (optional; default: 0) *see* TEXT
	72	Horiz. text justification type (optional; default: 0) *see* TEXT
	74	Vertical text justification type (optional; default: 0) *see* TEXT

Table B–3. Entity group codes by entity (continued)

Entity type	Group codes	Meaning
ATTDEF (continued)	11	Alignment point (optional: present only if 72 or 74 group is present and nonzero)
ATTRIB	10	Text start point
	40	Text height
	1	Value (string)
	2	Attribute tag (string)
	70	Attribute flags: 1 Attribute is invisible (does not display) 2 This is a constant Attribute 4 Verification is required on input of this Attribute 8 Attribute is preset (no prompt during insertion)
	73	Field length (optional; default: 0)
	50	Text rotation (optional; default: 0)
	41	Relative *X* scale factor (optional; default: 1)
	51	Oblique angle (optional; default: 0)
	7	Text style name (optional; default: STANDARD)
	71	Text-generation flags (optional; default: 0) *see* TEXT
	72	Horiz. text justification type (optional; default: 0) *see* TEXT
	74	Vertical text justification type (optional; default: 0) *see* TEXT
	11	Alignment point (optional: present only if 72 or 74 group is present and nonzero)
ARC	10	Centre
	40	Radius
	50	Start angle
	51	End angle
CIRCLE	10	Centre point
	40	Radius
DIMENSION	2	Name of pseudo-Block that contains the dimension picture
	3	Dimension style name
	10	Definition point
	11	Middle point of dimension text
	12	Insertion point for clones of a dimension (for Baseline and Continue)

Table B–3. Entity group codes by entity (continued)

Entity type	Group codes	Meaning
DIMENSION (continued)	70	Dimension type—these are integer codes, *not* bit-coded: 0 Rotated, horizontal, or vertical 1 Aligned 2 Angular 3 Diameter 4 Radius 5 Angular 3 point 6 Ordinate 64 *X*-type ordinate at the default location 192 *X*-type ordinate at a user-defined location
	1	Dimension text entered by the user (optional; default: the measurement)
	13	Definition point for linear and angular dimensions
	14	Definition point for linear and angular dimensions
	15	Definition point for diameter, radius, and angular dimensions
	16	Point defining dimension arc for angular dimensions
	40	Leader length for radius and diameter dimensions
	50	Angle of rotated, horizontal, or vertical linear dimensions
	51	Horizontal direction (optional)
	52	Extension line angle for oblique linear dimensions (optional)
	53	Rotation angle of dimension text (optional)
INSERT	66	Attributes-follow flag (optional; default: 0)
	2	Block name
	10	Insertion point
	41	*X* scale factor (optional; default: 1)
	42	*Y* scale factor (optional; default: 1)
	43	*Z* scale factor (optional; default: 1)
	50	Rotation angle (optional; default: 0)
	70	Column count (optional; default: 1)
	71	Row count (optional; default: 1)
	44	Column spacing (optional; default: 0)
	45	Row spacing (optional; default: 0)
LINE	10	Start point
	11	End point

Table B–3. Entity group codes by entity (continued)

Entity type	Group codes	Meaning
POINT	10	Point
	50	Angle of *X* axis for the UCS in effect when the Point was drawn (optional; default: 0) Used when PDMODE is nonzero
POLYLINE	66	Vertices-follow flag (always 1 for a Polyline)
	10	A "dummy" point; the *X* and *Y* coordinates are always 0, and the *Z* coordinate specifies the Polyline's elevation
	70	Polyline flag (optional; default: 0): 1 This is a closed Polyline (or a polygon mesh closed in the *M* direction) 2 Curve-fit vertices have been added 4 Spline-fit vertices have been added 8 This is a 3D Polyline 16 This is a 3D Polygon mesh 32 The polygon mesh is closed in the *N* direction 64 This Polyline is a polyface mesh 128 The linetype pattern is generated continuously around the vertices of this Polyline
	40	Default starting width (optional; default: 0)
	41	Default ending width (optional; default: 0)
	71	Polygon mesh *M* vertex count (optional; default: 0)
	72	Polygon mesh *N* vertex count (optional; default: 0)
	73	Smooth surface *M* density (optional; default: 0)
	74	Smooth surface *N* density (optional; default: 0)
	75	Curves and smooth surface type (optional; default: 0)—these are integer codes, *not* bit-coded: 0 No smooth surface fitted 5 Quadratic B-spline surface 6 Cubic B-spline surface 8 Bezier surface
SEQEND	–2	Name of entity that began the sequence
SHAPE	10	Insertion point
	40	Size
	2	Shape name
	50	Rotation angle (optional; default: 0)
	41	Relative *X*-scale factor (optional; default: 1)
	51	Oblique angle (optional; default: 0)

Table B–3. Entity group codes by entity (continued)

Entity type	Group codes	Meaning
SOLID	10	First corner
	11	Second corner
	12	Third corner
	13	Fourth corner (if only three corners entered, this equals the third corner)
TEXT	10	Insertion point
	40	Height
	1	Text value (the string itself)
	50	Rotation angle (optional; default: 0)
	41	Relative *X*-scale factor (optional; default: 0)
	51	Oblique angle (optional; default: 0)
	7	Text style name (optional; default: STANDARD)
	71	Text generation flags (optional; default: 0): 2 Text is backward (mirrored in *X)* 4 Text is upside down (mirrored in *Y)*
	72	Horizontal alignment (optional; default: 0)—these are integer codes, *not* bit-coded: 0 Left 1 Centre 2 Right 3 Aligned (if vertical alignment = 0) 4 Middle (if vertical alignment = 0) 5 Fit (if vertical alignment = 0)
	73	Vertical alignment (optional; default: 0)—these are integer codes, *not* bit-coded: 0 Baseline 1 Bottom 2 Middle 3 Top
	11	Alignment point (optional: present only if 72 or 73 group is present and nonzero)
TRACE	10	First corner
	11	Second corner
	12	Third corner
	13	Fourth corner
VERTEX	10	Location
	40	Starting width (optional; default: 0)
	41	Ending width (optional; default: 0)
	42	Bulge (optional; default: 0)

Table B–3. Entity group codes by entity (continued)

Entity type	Group codes	Meaning
VERTEX (continued)	70	Vertex flags (optional; default: 0): 1 Extra vertex created by curve-fitting 2 Curve-fit tangent defined for this vertex. A curve-fit tangent direction of 0 may be omitted from the DXF output, but is significant if this bit is set 4 *not used* 8 Spline vertex created by spline-fitting 16 Spline frame control point 32 3D Polyline vertex 64 3D Polygon mexh vertex 128 Polyface mesh vertex
	50	Curve fit tangent direction (optional)
VIEWPORT	10	Centre point
	40	Width in paper space units
	41	Height in paper space units
	69	Viewport ID (changes each time drawing is opened; never saved)
	68	Viewport status field
	1001	Application ID (`"ACAD"`). This begins a section of XDATA that describes the Viewport. User applications can't modify this data; see the *AutoCAD Customization Manual* for details

Block and Table Group Codes

The following table, B–4, shows the 70 group flag bit values that apply to all table entries:

Table B–4. Group 70 bit codes that apply to all table entries

Flag bit value	Meaning
16	If set, table entry is externally dependent on an Xref
32	If this bit and bit 16 are both set, the externally dependent Xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is for the benefit of AutoCAD commands; it can be ignored by most programs that read DXF files, and need not be set by programs that write DXF files)

The following table, B–5, shows the group codes for Block definitions and table entries (as they would be saved in the TABLES and BLOCKS sections of a DXF file; optional codes are shown in gray).

Table B–5. Block and table group codes by entity

Entity type	Group codes	Meaning
APPID	2	User-registered application name (for XDATA)
	70	Standard flag values
BLOCK	*Note:* A Block description also contains the standard entity groups shown in table B-2 *except* for the entity name (–1) group, which `ads_tblnext()` and `ads_tblsearch()` do not return	
	2	Block name
	70	Type flag: 1 This is an anonymous Block (generated by hatching, associative dimensioning, other internal operations, or an application) 2 This Block has Attributes 4 This Block is an external reference (Xref) 8 *not used* 16 This Block is externally dependent 32 This is a resolved external reference, or dependent of an external reference 64 This definition is referenced
	10	Base point
DIMSTYLE	2	Dimension style name
	70	Standard flag values
	3	DIMPOST
	4	DIMAPOST
	5	DIMBLK
	6	DIMBLK1
	7	DIMBLK2
	40	DIMSCALE
	41	DIMASZ
	42	DIMEXO
	43	DIMDLI
	44	DIMEXE
	45	DIMRND
	46	DIMDLE
	47	DIMTP
	48	DIMTM
	140	DIMTXT
	141	DIMCEN

Table B–5. Block and table group codes by entity (continued)

Entity type	Group codes	Meaning
DIMSTYLE	142	DIMTSZ
(continued)	143	DIMALTF
	144	DIMLFAC
	145	DIMTVP
	146	DIMTFAC
	147	DIMGAP
	71	DIMTOL
	72	DIMLIM
	73	DIMTIH
	74	DIMTOH
	75	DIMSE1
	76	DIMSE2
	77	DIMTAD
	78	DIMZIN
	170	DIMALT
	171	DIMALTD
	172	DIMTOFL
	173	DIMSAH
	174	DIMTIX
	175	DIMSOXD
	176	DIMCLRD
	177	DIMCLRE
	178	DIMCLRT
ENDBLK	(No groups)	End Block definition (appears *only* in BLOCKS table)
LAYER	2	Layer name
	70	Layer flags: 1 If set, layer is frozen 2 If set, layer is frozen by default in new Viewports 4 If set, layer is locked
	62	Colour
	6	Linetype

Table B–5. Block and table group codes by entity (continued)

Entity type	Group codes	Meaning
LTYPE	2	Linetype name
	70	Linetype flags
	3	Descriptive text for linetype
	72	Alignment code
	73	Number of dash length items
	40	Total pattern length
	49	Dash length (optional: can be repeated)
STYLE	2	Style name
	70	Style flags
	40	Fixed text height
	41	Width factor
	50	Oblique angle
	71	Text-generation flags: 2 Text is backward (mirrored in *X)* 4 Text is upside down (mirrored in *Y)*
	42	Last height used
	3	Primary font filename
	4	Big-font filename (empty string if none)
UCS	2	UCS name
	70	Standard flag values
	10	Origin in WCS
	11	*X* axis direction (in WCS)
	12	*Y* axis direction (in WCS)
VIEW	2	View name
	70	View flag: 1 If set, this View is a paper space view
	40	Height
	41	Width
	10	Centre point (a 2D point)
	11	View direction from target, in WCS
	12	Target point, in WCS
	42	Lens length
	43	Front clipping plane
	44	Back clipping plane

Table B–5. Block and table group codes by entity (continued)

Entity type	Group codes	Meaning
VIEW	50	Twist angle
(continued)	71	View mode
VPORT	1	Vport name (*might not be unique:* all Vports in the current configuration are named *ACTIVE, and the first *ACTIVE Vport in the table is the one currently displayed)
	70	Standard flag values
	10	Lower-left corner (a 2D point)
	11	Upper-right corner (a 2D point)
	12	Centre (a 2D point)
	13	Snap base point (a 2D point)
	14	Snap spacing (*X* and *Y*)
	15	Grid spacing (*X* and *Y*)
	16	Direction from target point
	17	Target point
	40	Height
	41	Aspect ratio
	42	Lens length
	43	Front clipping plane
	44	Back clipping plane
	50	Snap rotation angle
	51	Twist angle
	68	Status field
	69	ID
	71	View mode—same values as the VIEWMODE system variable
	72	Circle zoom percent
	73	Fast zoom setting
	74	UCSICON setting
	75	Snap on/off
	76	Grid on/off
	77	Snap style
	78	Snap isopair

Appendix C
Error Codes

The table in this appendix shows the values of error codes generated by AutoLISP. The AutoCAD system variable ERRNO is set to one of these values when an AutoLISP function call causes an error that AutoCAD detects. AutoLISP applications can inspect the current value of ERRNO with `(getvar "errno")`.

Note: The variable ERRNO is not always cleared to zero, so unless it is inspected immediately after an AutoLISP function has reported an error, the error its value indicates may be misleading. This variable is always cleared upon entry to the drawing editor.

Caution: The possible values of ERRNO, and their meanings, may change in future releases of AutoCAD.

Table C–1. On-line program error codes

Value	Meaning	AutoLISP Functions

1	Invalid symbol table name	`entmake`
		`entmod`
		`regapp`

2	Invalid entity or selection set name	Several functions[a] (see note that follows this table)
3	Exceeded maximum number of selection sets	`ssget`
4	Invalid selection set	`ssget`
5	Improper use of Block definition entity	Several functions[a]
6	Improper use of Xref entity	Several functions[a]
7	Entity selection: pick failed	`entsel` `nentsel`
8	End of entity file	`entnext` `entupd`
9	End of Block definition file	`entnext`
10	Failed to find last entity	`entlast`
11	Illegal attempt to delete Viewport entity	`entdel`
12	Operation not allowed during Pline	(Not currently used)
13	Invalid handle	`handent`

Table C–1. On-line program error codes (continued)

Value	Meaning	AutoLISP Functions
14	Handles not enabled	`handent`
15	Invalid arguments in coordinate transform request	`trans`
16	Invalid space in coordinate transform request	`trans`
17	Invalid use of deleted entity	`entmod` `trans`
18	Invalid table name	`tblnext` `tblsearch`
19	Invalid table function argument	`tblnext` `tblsearch`
20	Attempt to set a read-only variable	`setvar`
21	Zero value not allowed	`setvar`
22	Value out of range	`setvar`

23	Complex REGEN in progress	`entmake`
		`entmod`
		`entupd`

24	Attempt to change entity type	`entmake` `entmod`
25	Bad layer name	`entmake` `entmod`
26	Bad linetype name	`entmake` `entmod`
27	Bad color name	`entmake` `entmod`
28	Bad text style name	`entmake`
29	Bad shape name	`entmake`
30	Bad field for entity type	`entmake` `entmod`
31	Attempt to modify deleted entity	`entmod`
32	Attempt to modify Seqend subentity	`entmod`
33	Attempt to change handle	`entmod`
34	Attempt to modify Viewport visibility	`entmake` `entmod`
35	Entity on locked layer	`entmake` `entmod`
36	Bad entity type	`entmake`
37	Bad Pline entity	`entmake`
38	Incomplete complex entity in Block	`entmake`
39	Invalid Block name field	(Not currently used)

Table C–1. On-line program error codes (continued)

Value	Meaning	AutoLISP Functions
40	Duplicate Block flag fields	`entmake`
41	Duplicate Block name fields	`entmake`
42	Bad normal vector	`entmake`
43	Missing Block name	`entmake`
44	Missing Block flags	`entmake`
45	Invalid anonymous Block	`entmake`
46	Invalid Block definition entity	`entmake`
47	Mandatory field missing	`entmake`
48	Unrecognized extended data (XDATA) type	`entmake` `entmod`
49	Improper nesting of list in XDATA	`entmake` `entmod`
50	Improper location of APPID field	`entmake` `entmod`
51	Exceeded maximum XDATA size	`entmake` `entmod`
52	Entity selection: null response	`entsel` `nentsel`
53	Duplicate APPID	`entmake` `entmod`
54	Attempt to make or modify Viewport entity	`entmake` `entmod`
55	Attempt to make or modify an Xref, Xdef, or Xdep	`entmake` `entmod`
56	`ssget` filter: unexpected end of list	`ssget`
57	`ssget` filter: missing test operand	`ssget`
58	`ssget` filter: invalid opcode (–4) string	`ssget`
59	`ssget` filter: improper nesting or empty conditional clause	`ssget`
60	`ssget` filter: mismatched begin and end of conditional clause	`ssget`
61	`ssget` filter: wrong number of arguments in conditional clause (for `NOT` or `XOR`)	`ssget`
62	`ssget` filter: exceeded maximum nesting limit	`ssget`
63	`ssget` filter: invalid group code	`ssget`
64	`ssget` filter: invalid string test	`ssget`
65	`ssget` filter: invalid vector test	`ssget`

Table C–1. On-line program error codes (continued)

Value	Meaning	AutoLISP Functions
66	`ssget` filter: invalid real test	`ssget`
67	`ssget` filter: invalid integer test	`ssget`
68	Digitizer isn't a tablet	`tablet`
69	Tablet is not calibrated	`tablet`
70	Invalid arguments	`tablet`
71	ADS error: Unable to allocate new result buffer	
72	ADS error: Null pointer detected	
73	Can't open executable file	`xload`
74	Application is already loaded	`xload`
75	Maximum number of applications already loaded	`xload`
76	Unable to execute application	`xload`
77	Incompatible version number	`xload`
78	Unable to unload nested application	`xunload`
79	Application refused to unload	`xunload`
80	Application is not currently loaded	`xunload`
81	Not enough memory to load application	`xload`
82	ADS error: Invalid transformation matrix	
83	ADS error: Invalid symbol name	
84	ADS error: Invalid symbol value	
85	AutoLISP/ADS operation attempted and prohibited while a dialogue box was displayed.	

a. The error codes 2,5,and 6 can be reported by several functions, including **`entdel`**, **`entget`**, **`entmod`**, **`entnext`**, **`entupd`**, **`redraw`**, **`regapp`**, **`ssadd`**, **`ssdel`**, **`ssmemb`** ,**`trans`**, and `xdroom`.

Appendix D Error Messages

When AutoLISP detects an error condition, it cancels the function in progress and calls the user `*error*` function with a message indicating the type of error. If no user `*error*` function is defined (or if `*error*` is bound to nil), the standard error action is to display the message in this form:

```
error: message
```

followed by a function traceback. If a user-defined *error* function exists, it calls that function with *message* passed as the only argument.

User Program Errors

The following is a list of error messages you see from time to time while writing and debugging AutoLISP functions. Most of these messages indicate typical LISP programming errors, such as these:

- misspelled function or symbol names
- the wrong type or number of function arguments
- mismatched parentheses
- mismatched quotes (unterminated strings)
- failure to check for proper completion of a function before attempting to use its result

Although these messages usually indicate user programming errors, they may also arise due to programming errors (bugs) in AutoLISP itself. If you can't see anything wrong with your program, please fill out a Bug Report and send it to Autodesk.

arguments of a defun can't have the same name

A function defined with multiple arguments of the same name will fail with this message.

AutoCAD rejected function

The arguments passed to an AutoCAD function were invalid (as in `setvar` of a read-only system variable, or `tblnext` with an invalid table name), or the function itself is invalid in the current context. For instance, you cannot use a `getxxx` user input function inside the `command` function.

AutoLISP stack overflow

The AutoLISP stack storage space has been exceeded. This can be due to excessive function recursion or very large function argument lists.

bad argument type

A function was passed an incorrect type of argument. (For instance, you can't take the `strlen` of an integer.)

bad association list

The list supplied to the `assoc` function does not consist of *(key value)* lists.

bad conversion code

This indicates that an invalid space identifier was passed to the `trans` function.

bad ENTMOD list

The argument passed to `entmod` is not a proper entity data list (as returned by `entget`).

bad ENTMOD list value

One of the sublists in the association list passed to `entmod` contains an improper value.

bad formal argument list

When evaluating this function, AutoLISP detected an invalid formal argument list. Perhaps the function is not a function at all, but rather a data list.

bad function

The first element in the list is not a valid function name. Perhaps it is a variable name or a number. This message can also indicate that the named function is improperly defined—don't forget the required formal argument list.

bad function code

This indicates that a bad function identifier was passed to the `tablet` command.

bad grvecs list value

Something passed in a `grvecs` list isn't a 2D or 3D point.

bad grvecs matrix value

A matrix passed to `grvecs` is malformed, or contains the wrong data type (e.g., STR, SYM, etc.).

bad list

An improperly formed list was passed to a function. This can occur if a real number begins with a decimal point. You must use a leading zero in such a case.

bad list of points

Used by `ssget` if a null list or a list containing items other than points is sent along with a F, CP, or WP request. Also used by `grvecs` .

bad node

Invalid item type encountered by `type` function.

bad node type in list

Invalid item type encountered by `foreach` function.

bad point argument
bad point value

A poorly defined point (list of two reals) was passed to a function expecting a point. Be careful not to begin a real number with a decimal point; you must use a leading zero in such a case.

bad real number detected

An attempt was made to pass an invalid real number from AutoLISP to AutoCAD.

bad ssget list

The argument passed to (`ssget` "`X`") is not a proper entity data list (as returned by `entget`).

bad ssget list value

One of the sublists in the association list passed to (`ssget` "`X`") contains an improper value.

bad ssget mode string

This error is caused when `ssget` is passed an invalid string in the *mode* argument.

bad xdata list

This error is caused when `xdsize`, `ssget`, `entmod`, `entmake`, or `textbox` are passed a malformed extended entity data list.

base point is required

The `getcorner` function was called without the required base point argument.

Boole arg1 *0 or* 15

The first argument to the Boole function must be an integer between 0 and 15.

can't evaluate expression

This error can be caused by improper placement of a decimal point and other poorly formed expressions.

can't open (file) for input -- LOAD failed

The file named in the `load` function could not be found, or the user does not have read access to the file.

can't reenter AutoLISP

The AutoCAD/AutoLISP communication buffer is in use by an active function; no new function can be called until the active one is complete.

console break

The user entered Ctrl+C while a function was processing.

divide by zero

Division by zero is not allowed.

divide overflow

Division by a very small value has resulted in an invalid quotient.

exceeded maximum string length

A string passed to a function is greater than 132 characters.

extra right paren

One extra right parenthesis or more was encountered.

file not open

The file descriptor for the I/O operation is not that of an open file.

file read—insufficient string space

String space was exhausted while reading from a file. See "Memory Management" on page 177.

file size limit exceeded

A file has exceeded the operating system's file size limit.

floating-point exception

(UNIX-based systems only.) The operating system has detected an error in floating-point arithmetic.

function cancelled

The user entered Ctrl+C in response to an input prompt.

function undefined for argument

The argument passed to `log` or `sqrt` is out of range.

function undefined for real

A real number was passed as an argument to a function requiring an integer. For instance: `(lsh val 1.2)`.

grvecs missing endpoint

The vector list passed to `grvecs` is missing an endpoint.

illegal type in left

Means that the LSP file is not pure ASCII, but was saved by a word processing program and the file includes fomatting codes.

improper argument

Argument to `gcd` is negative or zero.

inappropriate object in function

An improperly constructed function has been detected by the `vmon` function pager.

incorrect number of arguments

The `quote` function expects exactly one argument, but some other number of arguments were supplied.

incorrect number of arguments to a function

The number of arguments to the user-defined function does not match the number of formal arguments specified in the Defun.

incorrect request for command list data

A `command` function was encountered but cannot be executed due to another active function.

input aborted

An error or premature end-of-file condition has been detected, causing termination of the file input.

insufficient node space

There is not enough heap space to accommodate the requested action. See "Memory Management" on page 177.

insufficient string space

There is not enough heap space to accommodate the specified text string. See "Memory Management" on page 177.

invalid argument

Improper argument type, or argument out of range.

invalid argument list

A corrupted argument list was passed to a function.

invalid character

An expression contains an improper character.

invalid dotted pair

Dotted pairs are lists containing two elements separated by a *space-period-space* construction. You might get this error message if you begin a real number with a decimal point; you must use a leading zero in such a case.

invalid integer value

A number smaller than the smallest integer, or larger than the largest integer, was encountered.

LISPSTACK overflow

The AutoLISP stack storage space has been exceeded. This can be due to excessive function recursion or very large function argument lists.

malformed list

A list being read from a file has ended prematurely. The most common cause is a mismatch in the pairings of opening and closing parenthesis or quotation marks.

malformed string

A string being read from a file has ended prematurely.

misplaced dot

This can occur if a real number begins with a decimal point; you must use a leading zero in such a case.

null function

An attempt was made to evaluate a function that has a nil definition.

quit/exit abort

This is the result of a call to the `quit` or `exit` function.

string too long

A string passed to `setvar` is too long.

too few arguments

Too few arguments were passed to a built-in function.

too few arguments to grvecs

Not enough arguments were passed to `grvecs`.

too many arguments

Too many arguments were passed to a built-in function.

Internal Errors

You should rarely, if ever, encounter the following error messages. They tend to indicate internal errors in the AutoLISP program itself, and you should report them to Autodesk on a Bug Report form.

bad argument to system call

(UNIX-based systems only.) The operating system has detected a bad system call generated by AutoLISP.

bus error

(UNIX-based systems only.) The operating system has detected a bus error.

hangup

(UNIX-based systems only.) The operating system has detected a *hangup* signal.

illegal instruction

(UNIX-based systems only.) The operating system has detected an invalid machine instruction.

segmentation violation

(UNIX-based systems only.) The operating system has detected an attempt to address outside this process's memory area.

unexpected signal *nnn*

(UNIX-based systems only.) An unexpected signal was received from the operating system.

Appendix E Tutorial

One of the most powerful capabilities for extending AutoCAD is the AutoLISP language. This facility, provided with AutoCAD, is an implementation of the LISP programming language coupled to AutoCAD. By writing programs in AutoLISP, you can add commands to AutoCAD and modify AutoCAD much like our own software developers.

This tutorial shows you how to add a new command to AutoCAD. It explains how AutoLISP works and shows you how to put its power to work for you. The command you're going to develop is oriented to landscape architecture, but the concepts you'll learn are relevant regardless of your application area.

We assume that you're a reasonably proficient AutoCAD user—that is, you know the AutoCAD commands and the general concepts of AutoCAD. We also assume you have access to a text editor that can construct ASCII files. You'll be writing a program here—use your text editor to do what the tutorial asks.

You will use numerous AutoLISP functions in this tutorial; chapter 4 of this manual contains a complete explanation of all of these functions.

The Goal

Our goal is to develop a new command for AutoCAD that draws a garden path and fills it with circular concrete tiles. Your new command will have the prompt sequence:

```
Command: path
Start point of path: Locate the start point.
Endpoint of path: Locate the endpoint.
Half width of path: Enter a number.
Radius of tiles: Enter a number.
Spacing between tiles: Enter a number.
```

You enter the *start point* and *endpoint* to specify the centre line of a path. Next, you enter the half width of the path and the radius of the circular tiles. Finally, you enter the spacing between the tiles. You are specifying the half width of the path rather than the full width because it is easier to visualize the half width with the line that rubber bands from the start point.

Getting Started

You'll develop this application as most are done, from the inside out (or bottom up). You'll be making heavy use of angles in this application. AutoLISP, like many programming languages, specifies angles in radians. Radians measure angles from zero to 2 * π (PI). Since most users think of angles in terms of degrees, you'll define a *function* that converts degrees to radians. Using your text editor, create a file called *gp.lsp*. Enter the following program:

```
; Convert angle in degrees to radians

(defun dtr (a)
   (* pi (/ a 180.0))
)
```

Now consider what this does. You're defining a function using the **defun** function in AutoLISP. The function is called **dtr** (short for *degrees to radians*). It takes one argument, a, the angle in degrees. Its result is this expression:

```
π * (a / 180.0)
```

It is expressed in LISP notation that you can read as *the product of PI multiplied by the quotient of* A *divided by 180.0.* π is predefined by AutoLISP as 3.14159.... The line beginning with a semicolon is a *comment*—AutoLISP ignores all text on a line after a semicolon.

Save the file to disk, and then bring up AutoCAD on a new drawing (the name doesn't matter as we won't be saving the drawing). When the AutoCAD Command: prompt appears, load the function by entering

```
Command: (load "gp")
```

AutoLISP loads your function, echoing its name DTR (it is assumed that *gp.lsp* is in the AutoCAD search path). From now on in this document when we say *bring up AutoCAD and load the program,* we mean the sequence just described.

Now you'll test the function by executing it with various values. From the definition of radians above, zero degrees should equal zero radians, so enter this:

```
Command: (dtr 0)
```

Entering a line that begins with a left parenthesis makes AutoCAD pass the line to AutoLISP for evaluation. In this case, we are evaluating the **dtr** function we just defined and passing it an argument of zero. After evaluating the function, AutoCAD prints the result, so this input should generate this reply:

```
0.0
```

Now try 180 degrees. If you enter

```
Command: (dtr 180)
```

you see this response:

```
3.14159
```

This indicates that 180 degrees is equal to π radians. If you examine the function, you see that this is how we defined it.

At this point you should QUIT AutoCAD and return to your text editor.

Getting Input

The "garden path" command will ask the user where to draw the path, how wide to make it, how large the concrete tiles are, and how closely to space them. You'll define a function that asks the user for all of these items and then computes various numbers to use in the rest of the command.

Using your text editor, add the following lines to *gp.lsp* (a vertical bar is shown in this manual to indicate the lines you add).

Note: If you are working on a DOS computer, you can use the SHELL command to escape to the operating system from the AutoCAD drawing, thereby letting you use your word processor to edit the file without exiting AutoCAD.

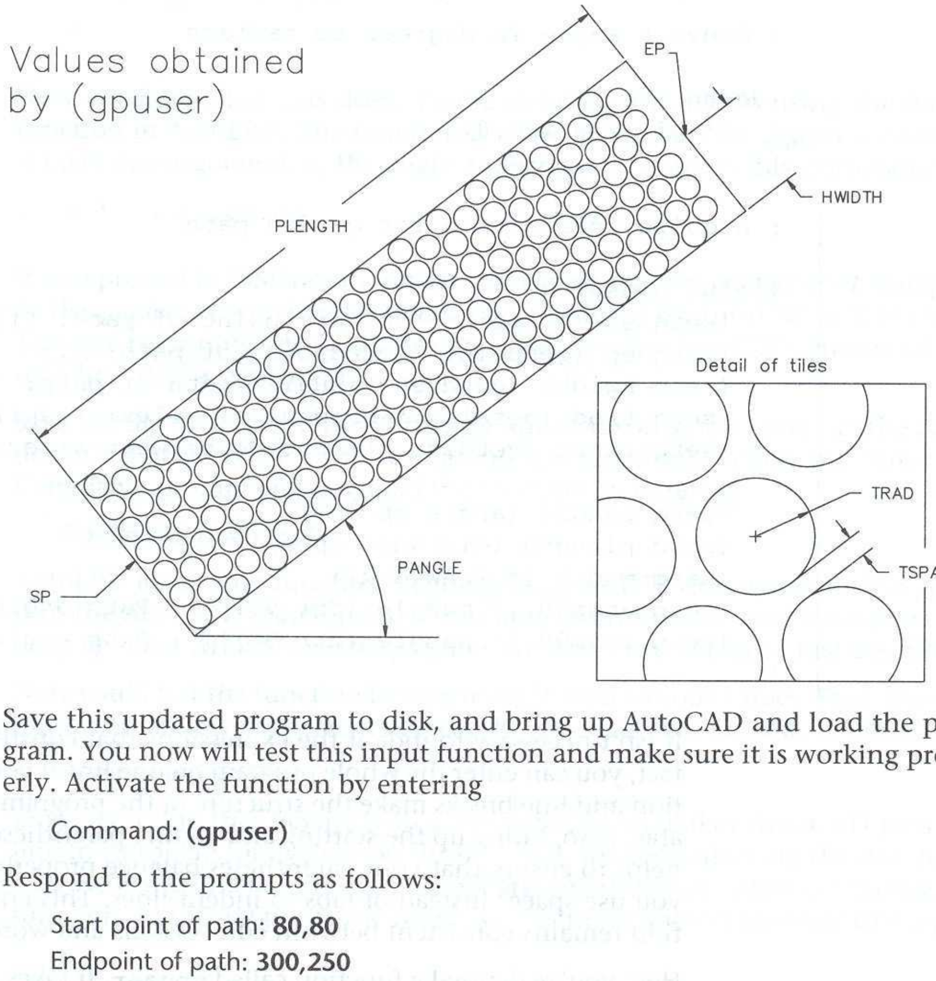
```
; Convert angle in degrees to radians

(defun dtr (a)
   (* pi (/ a 180.0))
)

; Acquire information for garden path

(defun gpuser ()
   (setq sp (getpoint "\nStart point of path: "))
   (setq ep (getpoint "\nEndpoint of path: "))
   (setq hwidth (getdist "\nHalf width of path: " sp))
   (setq trad (getdist "\nRadius of tiles: " sp))
   (setq tspac (getdist "\nSpacing between tiles: " sp))

   (setq pangle (angle sp ep))
   (setq plength (distance sp ep))
   (setq width (* 2 hwidth))
   (setq angp90 (+ pangle (dtr 90))) ; Path angle + 90 deg
   (setq angm90 (- pangle (dtr 90))) ; Path angle - 90 deg
)
```

It isn't necessary to indent the expressions that constitute your functions. In fact, you can enter the whole program on one line if you like. However, indention and line breaks make the structure of the program clearer and more readable. Also, lining up the starting and ending parentheses of major expressions helps to ensure that your parentheses balance properly. We recommend that you use spaces instead of tabs to indent lines. This ensures that the indentation remains consistent between edit sessions and word processors.

Here you've defined a function called **`gpuser`**. It takes no arguments and asks the user for all of the desired items. The **`setq`** function sets an AutoLISP variable to a specific value. The first **`setq`** sets variable `sp` (start point) to the result of the **`getpoint`** function. The **`getpoint`** function obtains a point from the user. The string specifies the prompt AutoCAD uses to obtain the point. The `\n` causes the prompt to appear on a new line. We use the **`getdist`** function to obtain the half width of the path, the tile radius, and the spacing between the tiles. The second argument to the **`getdist`** function, `sp`, specifies the *base point* for the distance. This makes the distance, if specified by a point in AutoCAD, relative to the starting point of the path, and attaches a rubber-band line to that point.

After the input is obtained from the user, several commonly used variables are computed. The `pangle` variable is set to the angle from the start point to the endpoint of the path. The **`angle`** function returns this angle given two points. The `plength` variable is set to the length of the path. The **`distance`** function calculates a distance given two points. Since you specified the half width of the path, you calculate the width as twice this. Finally, you calculate and save the angle of the path plus and minus 90 degrees in `angp90` and `angm90` respectively (since angles within AutoLISP are in radians, you used the **`dtr`** function to convert degrees to radians before these calculations).

The following illustration shows how the variables obtained by **`gpuser`** specify the geometry of the path.



Save this updated program to disk, and bring up AutoCAD and load the program. You now will test this input function and make sure it is working properly. Activate the function by entering

Command: **(gpuser)**

Respond to the prompts as follows:

Start point of path: **80,80**
Endpoint of path: **300,250**
Half width of path: **60**
Radius of tiles: **6**
Spacing between tiles: **3**

The **`gpuser`** function uses your replies to compute the additional variables it needs and displays the result of its last computation (in this case, –0.912908, the value of `angm90` in radians). You can dump out all the variables set by the **`gpuser`** function by entering their names preceded by an exclamation point

(!). This causes AutoCAD to evaluate the variable and print the result. If you enter the following commands, you should receive the indicated results:

```
Command: !sp
(80.0 80.0 0.0)
Command: !ep
(300.0 250.0 0.0)
Command: !hwidth
60.0
Command: !width
120.0
Command: !trad
6.0
Command: !tspac
3.0
Command: !pangle
0.657889
Command: !plength
278.029
Command: !angp90
2.22868
Command: !angm90
-0.912908
```

The `sp` and `ep` variables are returned as 3D points (*X*, *Y*, and *Z*); ignore the *Z* component in this exercise.

Also, `pangle`, `angp90`, and `angm90` are represented in radians. After verifying these values, quit AutoCAD and return to your text editor on *gp.lsp*.

Getting Oriented

Now that you've asked the user for the location of the path, you can draw its outline. Add the lines indicated with the character "|" to your *gp.lsp* file.

```
; Convert angle in degrees to radians

(defun dtr (a)
   (* pi (/ a 180.0))
)

 ; Acquire information for garden path

(defun gpuser ()
   (setq sp (getpoint "\nStart point of path: "))
   (setq ep (getpoint "\nEndpoint of path: "))
   (setq hwidth (getdist "\nHalf width of path: " sp))
   (setq trad (getdist "\nRadius of tiles: " sp))
   (setq tspac (getdist "\nSpacing between tiles: " sp))

   (setq pangle (angle sp ep))
   (setq plength (distance sp ep))
   (setq width (*2 hwidth))
   (setq angp90 (+ pangle (dtr 90))) ; Path angle + 90 deg
   (setq angm90 (- pangle (dtr 90))) ; Path angle - 90 deg
)

; Draw outline of path

(defun drawout ()
   (command "pline"
      (setq p (polar sp angm90 hwidth))
      (setq p (polar p pangle plength))
      (setq p (polar p angp90 width))
      (polar p (+ pangle (dtr 180)) plength)
      "close"
   )
)
```

This addition defines a function called **drawout**. This function uses the starting point, angle, and length of the path obtained by the **gpuser** function, and draws the outline of the path as a Polyline. The **drawout** function uses the **command** function to submit commands and data to AutoCAD. The **command** function is how AutoLISP functions submit commands to be executed by AutoCAD. The **command** function takes any number of arguments and submits each to AutoCAD. In this case, you feed the command PLINE to AutoCAD, activating its Polyline command and then supply the four corners of the path. Each corner is developed using the **polar** function, and stored in the temporary variable p. The **polar** function takes a point as its first argument and an angle and distance supplied by its second and third arguments and returns a point the specified distance and angle from the original point. In this case you calculate the four points bounding the path geometrically from the start point of the path. You complete the command by sending the string *close* to the PLINE command, which causes it to draw the fourth side of the path and return to the AutoCAD Command: prompt.

To test this function, save the updated gp.lsp, bring up AutoCAD on a new drawing, and load the AutoLISP file as before. Activate the user input function as before:

```
Command: (gpuser)
```

and supply the values as in the last step. Then test the new **drawout** function by invoking it:

```
Command: (drawout)
```

Your function supplies the commands to AutoCAD to draw the border for the path, and the border appears on the screen. After testing the function, quit AutoCAD.

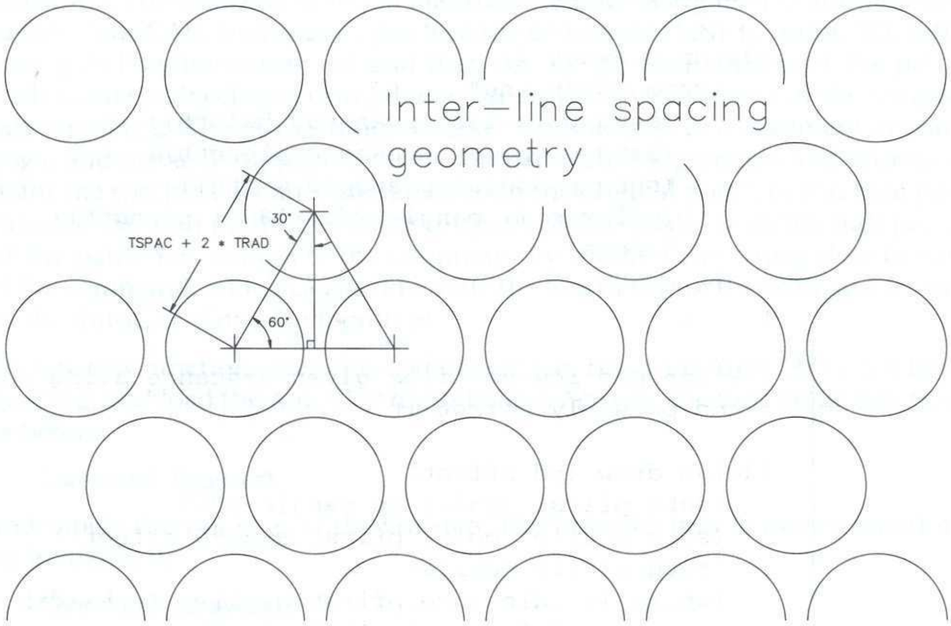
Drawing the Tiles

Now that you've developed and tested the user input function and the function that draws the border, you're ready to fill the path with the circular tiles. This requires some geometry. Bring up your text editor and add the indicated new functions to your program:

```
; Convert angle in degrees to radians

(defun dtr (a)
   (* pi (/ a 180.0))
)

; Acquire information for garden path

(defun gpuser ()
   (setq sp (getpoint "\nStart point of path: "))
   (setq ep (getpoint "\nEndpoint of path: "))
   (setq hwidth (getdist "\nHalf width of path: " sp))
   (setq trad (getdist "\nRadius of tiles: " sp))
   (setq tspac (getdist "\nSpacing between tiles: " sp))

   (setq pangle (angle sp ep))
   (setq plength (distance sp ep))
   (setq width (* 2 hwidth))
   (setq angp90 (+ pangle (dtr 90))) ; Path angle + 90 deg
   (setq angm90 (- pangle (dtr 90))) ; Path angle - 90 deg
)

; Draw outline of path

(defun drawout ()
   (command "pline"
      (setq p (polar sp angm90 hwidth))
      (setq p (polar p pangle plength))
      (setq p (polar p angp90 width))
      (polar p (+ pangle (dtr 180)) plength)
      "close"
   )
)

; Place one row of tiles given distance along path
; and possibly offset it

(defun drow (pd offset)
   (setq pfirst (polar sp pangle pd))
   (setq pctile (polar pfirst angp90 offset))
   (setq p1tile pctile)
   (while (< (distance pfirst p1tile) (- hwidth trad))
      (command "circle" p1tile trad)
      (setq p1tile (polar p1tile angp90 (+ tspac trad trad)))
   )
   (setq p1tile (polar pctile angm90 (+ tspac trad trad)))
   (while (< (distance pfirst p1tile) (- hwidth trad))
      (command "circle" p1tile trad)
      (setq p1tile (polar p1tile angm90 (+ tspac trad trad)))
   )
)
```

```
; Draw the rows of tiles

(defun drawtiles ()
   (setq pdist (+ trad tspac))
   (setq off 0.0)
   (while (<= pdist (- plength trad))
      (drow pdist off)
      (setq pdist (+ pdist (* (+ tspac trad trad) (sin (dtr 60)))))
      (if (= off 0.0)
         (setq off (* (+ tspac trad trad) (cos (dtr 60))))
         (setq off 0.0)
      )
   )
)
```

To understand how these functions work, refer to the following illustration. The function **drow** draws a row of tiles at a given distance along the path specified by its first argument, offsetting the row perpendicular to the path by a distance given by its second argument. You want to offset the tiles on alternate rows to cover more space with the tiles and to make a more pleasing arrangement.



The **drow** function finds the location for the first tile in the row by using **polar** to move along the path by the distance given by the first argument, and then **polar** again to move perpendicular to the path for the offset. It then uses the **while** function to continue to draw circles until the edge of the path is encountered. The **setq** at the end of the **while** loop moves on to the next tile location by spacing a distance of two tile radii and one intertile spacing.

A second **while** loop then draws the tiles in the row in the other direction until the other edge of the path is encountered.

The **drawtiles** function calls **drow** repeatedly to draw all the rows of tiles. Its **while** loop steps along the path calling **drow** for each row. Tiles in adjacent rows form equilateral triangles as shown in the illustration. The edges of these triangles are equal to twice the tile radius plus the spacing between the tiles.

Therefore, by trigonometry, the distance along the path between rows is the sine of 60 degrees multiplied by this quantity, and the offset for odd rows is the cosine of 60 degrees multiplied by this quantity.

The **if** function is used in **drawtiles** to offset every other row. The **if** function tests its first argument and executes the second argument if it is true and the third argument otherwise. In this case, if OFF is equal to zero, set it to the spacing between centres of tiles multiplied by the cosine of 60 degrees as explained above. If OFF is nonzero, we set it to zero. This alternates the offset on the rows as we want.

To test this function, save the file, and then bring up AutoCAD and load the program. Enter

Command: **(gpuser)**

and supply the path information as before. Enter

Command: **(drawout)**

and the outline should be drawn as before. Finally, enter

Command: **(drawtiles)**

and all of the tiles should be drawn within the border.

Adding the Command to AutoCAD

Finally, you're ready to combine the pieces into an AutoCAD command. If you define a function in AutoLISP with the name **C:*XXX***, entering ***XXX*** (assuming that ***XXX*** is not an AutoCAD command), invokes the function. To finish implementing the Path command, define a function **C:PATH**, which lets you enter **path** after loading *gp.lsp* to execute the garden path command.

Use your text editor to add the indicated lines to *gp.lsp*, and then bring up AutoCAD and load the program.

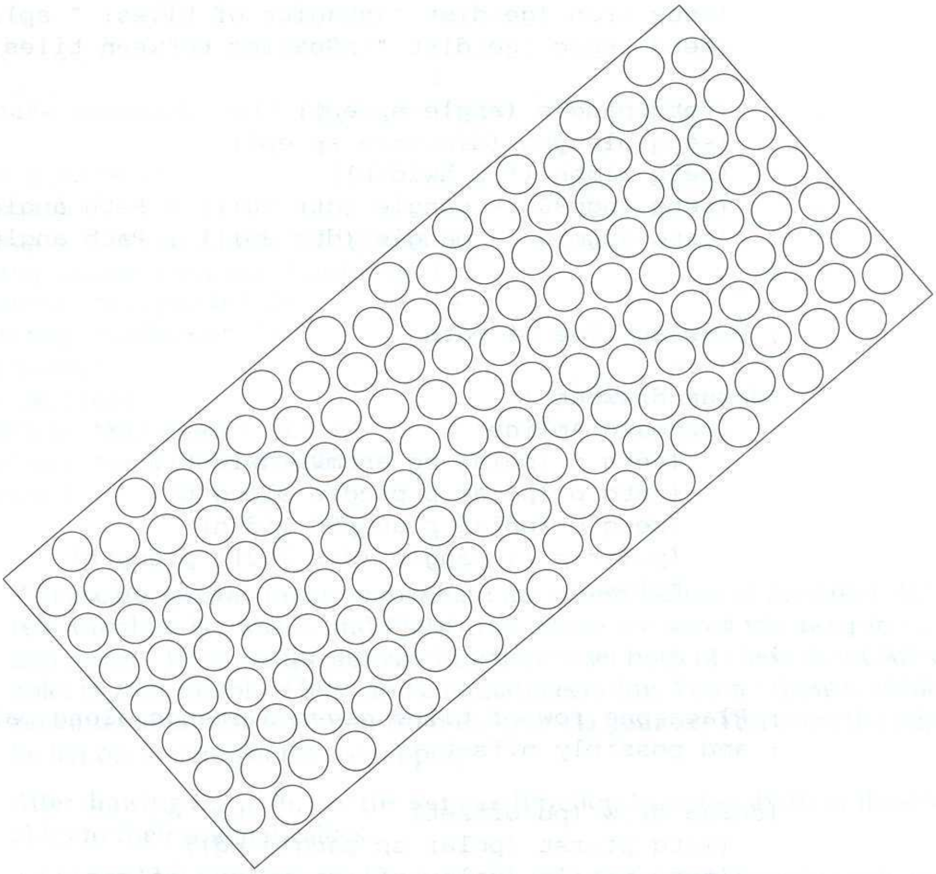
```
; Convert angle in degrees to radians

(defun dtr (a)
   (* pi (/ a 180.0))
)

; Acquire information for garden path

(defun gpuser ()
   (setq sp (getpoint "\nStart point of path: "))
   (setq ep (getpoint "\nEndpoint of path: "))
   (setq hwidth (getdist "\nHalf width of path: " sp))
   (setq trad (getdist "\nRadius of tiles: " sp))
   (setq tspac (getdist "\nSpacing between tiles: " sp))

   (setq pangle (angle sp ep))
   (setq plength (distance sp ep))
   (setq width (* 2 hwidth))
   (setq angp90 (+ pangle (dtr 90))) ; Path angle + 90 deg
   (setq angm90 (- pangle (dtr 90))) ; Path angle - 90 deg
)
```

```
; Draw outline of path

(defun drawout ()
   (command "pline"
      (setq p (polar sp angm90 hwidth))
      (setq p (polar p pangle plength))
      (setq p (polar p angp90 width))
      (polar p (+ pangle (dtr 180)) plength)
      "close"
   )
)

; Place one row of tiles given distance along path
; and possibly offset it

(defun drow (pd offset)
   (setq pfirst (polar sp pangle pd))
   (setq pctile (polar pfirst angp90 offset))
   (setq p1tile pctile)
   (while (< (distance pfirst p1tile) (- hwidth trad))
      (command "circle" p1tile trad)
      (setq p1tile (polar p1tile angp90 (+ tspac trad trad)))
   )
   (setq p1tile (polar pctile angm90 (+ tspac trad trad)))
   (while (< (distance pfirst p1tile) (- hwidth trad))
      (command "circle" p1tile trad)
      (setq p1tile (polar p1tile angm90 (+ tspac trad trad)))
   )
)

; Draw the rows of tiles

(defun drawtiles ()
   (setq pdist (+ trad tspac))
   (setq off 0.0)
   (while (<= pdist (- plength trad))
      (drow pdist off)
      (setq pdist (+ pdist (* (+ tspac trad trad) (sin (dtr 60)))))
      (if (= off 0.0)
         (setq off (* (+ tspac trad trad) (cos (dtr 60))))
         (setq off 0.0)
      )
   )
)

; Execute command, calling constituent functions

(defun C:PATH ()
   (gpuser)
   (drawout)
   (drawtiles)
)
```

By adding a function called **C:PATH** we've added a PATH command to AutoCAD. You can test the command by entering

```
Command: path
Start point of path: 80,80
Endpoint of path: 300,250
Half width of path: 60
Radius of tiles: 6
Spacing between tiles: 3
```

This should draw a garden path as shown in the following figure.



Finishing Up

As the PATH command executes, all the commands it submits to AutoCAD are echoed to the command/prompt area, and all the points it selects are flagged on the screen with small crosses (blips). Once a command function is debugged, you can turn off this output to make the AutoLISP-implemented command appear exactly like an AutoCAD command. Add the lines indicated with the character "|" to *gp.lsp* to suppress command echoing and blips.

```
; Convert angle in degrees to radians

(defun dtr (a)
   (* pi (/ a 180.0))
)

; Acquire information for garden path

(defun gpuser ()
   (setq sp (getpoint "\nStart point of path: "))
   (setq ep (getpoint "\nEndpoint of path: "))
   (setq hwidth (getdist "\nHalf width of path: " sp))
   (setq trad (getdist "\nRadius of tiles: " sp))
   (setq tspac (getdist "\nSpacing between tiles: " sp))

   (setq pangle (angle sp ep))
   (setq plength (distance sp ep))
   (setq width (* 2 hwidth))
   (setq angp90 (+ pangle (dtr 90))) ; Path angle + 90 deg
   (setq angm90 (- pangle (dtr 90))) ; Path angle - 90 deg
)

; Draw outline of path

(defun drawout ()
   (command "pline"
      (setq p (polar sp angm90 hwidth))
      (setq p (polar p pangle plength))
      (setq p (polar p angp90 width))
      (polar p (+ pangle (dtr 180)) plength)
      "close"
   )
)

; Place one row of tiles given distance along path
; and possibly offset it

(defun drow (pd offset)
   (setq pfirst (polar sp pangle pd))
   (setq pctile (polar pfirst angp90 offset))
   (setq p1tile pctile)
   (while (< (distance pfirst p1tile) (- hwidth trad))
      (command "circle" p1tile trad)
      (setq p1tile (polar p1tile angp90 (+ tspac trad trad)))
   )
   (setq p1tile (polar pctile angm90 (+ tspac trad trad)))
   (while (< (distance pfirst p1tile) (- hwidth trad))
      (command "circle" p1tile trad)
      (setq p1tile (polar p1tile angm90 (+ tspac trad trad)))
   )
)
```

```
; Draw the rows of tiles

(defun drawtiles ()
   (setq pdist (+ trad tspac))
   (setq off 0.0)
   (while (<= pdist (- plength trad))
      (drow pdist off)
      (setq pdist (+ pdist (* (+ tspac trad trad) (sin (dtr 60)))))
      (if (= off 0.0)
         (setq off (* (+ tspac trad trad) (cos (dtr 60))))
         (setq off 0.0)
      )
   )
)

; Execute command, calling constituent functions

(defun C:PATH ()
   (gpuser)
   (setq sblip (getvar "blipmode"))
   (setq scmde (getvar "cmdecho"))
   (setvar "blipmode" 0)
   (setvar "cmdecho" 0)
   (drawout)
   (drawtiles)
   (setvar "blipmode" sblip)
   (setvar "cmdecho" scmde)
   (princ)
)
```

You use the **getvar** function to obtain the current values of the AutoCAD system variables BLIPMODE and CMDECHO. These are saved via **setq** in sblip and scmde. Then use the **setvar** function to set both of these AutoCAD variables to zero, disabling blips and command echoing. You set these variables to zero after obtaining the input from the user via **gpuser**. You want the blips to be left on to confirm the user input.

After drawing the path, use the **setvar** function to restore both of these variables to their original values.

Adding a final call to the **princ** function lets the **C:PATH** function *exit quietly*. AutoLISP functions always return the value of the last function call; in this case, a 1 or 0 is returned if the final call to **princ** is omitted.

Save the file, bring up AutoCAD and try the PATH command now. Play around with it, specifying the various inputs from the pointer as well as the keyboard.

Adding a Dialogue Box Interface

The Dialogue Control Language (DCL) lets you add user definable dialogue boxes to your AutoLISP programs.

Your new PATH command accepts its input at the command line. You can easily add a dialogue box interface to this command by using the dialogue box functions described in the section "Programmable Dialogue Box Functions" on page 85 and by creating a *.dcl* file that contains the DCL description of the dialogue box. The AutoLISP dialogue box functions and DCL are described fully in chapter 9 of the *AutoCAD Customization Manual*.

Dialogue boxes are very useful for letting users choose between a number of options, define sizes, and specify amounts before making the final decision to execute a command. Currently this program has very few options which would make for a boring dialogue box; we will take this opportunity to add a few more features.

A new feature we will add to the **C:PATH** function will let you specify the shape of the tiles in the path. We will also add an error handling function.

You should start by copying the finished version of *gp.lsp* to another file, *ddgp.lsp* (most of the dialogue box interface AutoLISP programs supplied with AutoCAD are named with this *ddxxx.lsp* format). We will also be creating a new ASCII file *ddgp.dcl* that contains the DCL description of the dialogue box.

The DCL File—*ddgp.dcl*

The dialogue box you create will contain two radio buttons (if you select one, the other is deselected ... you know, like on a car radio) for choosing the tile shape: circle or polygon. It will also contain three edit boxes for entering the numeric values: radius of the tile, spacing between the tiles, and the number of sides (which is only available if the radio button Polygon is selected).

Without explaining in great detail the mechanics of DCL and the associated AutoLISP functions, we will outline the steps required to add a simple dialogue box to the **C:PATH** function. The following code should be placed in the file *ddgp.dcl*; this is the DCL file.

```
/* DDGP.DCL - DCL file for DDGP.LSP */

gp_box1 : dialog {
   label = "Garden Path Tile Specifications";
   : boxed_radio_row {
      label = "Tile Shape";
      : radio_button {
         label = "Polygon";
         mnemonic = "P";
         key = "gp_poly";
      }
      : radio_button {
         label= "Circle";
         mnemonic = "C";
         key = "gp_circ";
         value = "1";
      }
   }
   : edit_box {
      label = "Radius of tile";
      mnemonic = "R";
      key = "gp_trad";
      edit_width = 6;
   }
```

Defines the Polygon radio button

Defines the Circle radio button

Defines the radio button area

Defines the Radius of tile edit box

```
        : edit_box {
           label = "Spacing between tiles";
           mnemonic = "S";
           key = "gp_spac";
           edit_width = 6;
        }
        : edit_box {
           label = "Number of sides";
           mnemonic = "N";
           key = "gp_side";
           edit_width = 4;
        }
        : row {
          : spacer { width = 1; }
          : button {
             label = "OK";
             key = "accept";
             width = 8;
             fixed_width = true;
          }
          : button {
             label = "Cancel";
             is_cancel = true;
             key = "cancel";
             width = 8;
             fixed_width = true;
          }
          : spacer { width = 1;}
        }
    }
```

Defines the Spacing between tiles edit box

Defines the Number of sides edit box

Defines the OK button

Defines the Cancel button

Defines the OK/Cancel button row

Dialogue Box Functions in AutoLISP—*ddgp.lsp*

Now that you have created the DCL file, you can edit the necessary lines in the AutoLISP file *ddgp.lsp* you created from *gp.lsp*.

As with the DCL file, we won't explain here what each function does; the function names indicate the action each performs. The function **`load_dialog`** does just what it says: it loads a dialogue. The function **`set_tile`** sets a specified tile's initial value (each button, edit box, and so on is called a tile) to a string value, just as **`action_tile`** determines the action that will be taken when that tile is activated (edited, selected, pushed, and so on).

The new lines of code are marked with the character “ | ” as before. Some of the old lines need to be commented out or removed from this new file; these lines are in **`bold`**, marked with two semicolons at the beginning of the line (**`;;`**), and have the note **`;<-REMOVE`** at the end of the line.

```
;;; DDGP.LSP - the good old Garden Path with a new twist.

; Convert angle in degrees to radians

(defun dtr (a)
 (* pi (/ a 180.0))
)
```

```
; Acquire information for garden path

(defun gpuser ()
   (setq sp (getpoint "\nStart point of path: "))
   (setq ep (getpoint sp "\nEndpoint of path: ")); <<---ADDED 'sp'
   (setq hwidth (getdist "\nHalf width of path: " sp))
;; (setq trad (getdist "\nRadius of tiles: " sp))          ;<-REMOVE
;; (setq tspac (getdist "\nSpacing between tiles: " sp)) ;<-REMOVE
   (setq pangle (angle sp ep))
   (setq plength (distance sp ep))
   (setq width (* 2 hwidth))
   (setq angp90 (+ pangle (dtr 90)))
   (setq angm90 (- pangle (dtr 90)))
)

; Draw outline of path

(defun drawout ()
   (command "pline"
            (setq p (polar sp angm90 hwidth))
            (setq p (polar p pangle plength))
            (setq p (polar p angp90 width))
            (polar p (+ pangle (dtr 180)) plength)
            "close"
   )
)

; Call dialogue box to set tile specifications

(defun gp_dialog ()
   (setq tshape "Circle"
         trad 60
         tspac 6
         tsides 8 )
   (setq dcl_id (load_dialog "ddgp.dcl"))
   (if (not (new_dialog "gp_box1" dcl_id))(exit))
   (set_tile "gp_trad" "60")
   (set_tile "gp_spac" "6")
   (mode_tile "gp_side" 1)
   (set_tile "gp_side" "8")
   (action_tile "gp_circ"
      "(setq tshape \"Circle\")(mode_tile \"gp_side\" 1)")
   (action_tile "gp_poly"
      "(setq tshape \"Polygon\")(mode_tile \"gp_side\" 0)")
   (action_tile "cancel" "(done_dialog)(setq gperr \"\")(exit)")
   (action_tile "accept"
      (strcat
         "(progn (setq trad (atof (get_tile \"gp_trad\")))"
         "(setq tspac (atof (get_tile \"gp_spac\")))"
         "(setq tsides (atoi (get_tile \"gp_side\")))"
         "(done_dialog))"
      )
   )
   (start_dialog)
   (unload_dialog dcl_id)
   (if (= tshape "Circle")
      (defun gp_tile () (command "Circle" p1tile trad))
      (defun gp_tile () (command "Polygon" tsides p1tile "" trad))
   )
)

; Define error handler

(defun gp_err (msg)
   (setq *error* olderr)
   (if (not gperr)
      (princ (strcat "\nGarden path error: " msg))
      (princ gperr)
   )
   (if sblip (setvar "blipmode" sblip))
   (if scmde (setvar "cmdecho" scmde))
   (princ)
)

; Place one row of tiles given distance along path
; and possibly offset it

(defun drow (pd offset)
   (setvar "snapang" pangle)
   (setq pfirst (polar sp pangle pd))
   (setq pctile (polar pfirst angp90 offset))
   (setq p1tile pctile)
   (while (< (distance pfirst p1tile) (- hwidth trad))
      (gp_tile)
;;    (command "circle" p1tile trad)                     ;<-REMOVE
      (setq p1tile (polar p1tile angp90 (+ tspac trad trad)))
   )
   (setq p1tile (polar pctile angm90 (+ tspac trad trad)))
   (while (< (distance pfirst p1tile) (- hwidth trad))
      (gp_tile)
;;    (command "circle" p1tile trad)                     ;<-REMOVE
      (setq p1tile (polar p1tile angm90 (+ tspac trad trad)))

)

; Draw the rows of tiles

(defun drawtiles ()
   (setq pdist (+ trad tspac))
   (setq off 0.0)
   (while (<= pdist (- plength trad))
      (drow pdist off)
      (setq pdist (+ pdist (* (+ tspac trad trad) (sin (dtr 60)))))
      (if (= off 0.0)
         (setq off (* (+ tspac trad trad) (cos (dtr 60))))
      (setq off 0.0)
      )
   )
)
```

```
; Execute command, calling constituent functions

(defun C:DDPATH ()
   (setq olderr *error*
         *error* gp_err
         sblip nil
         scmde nil
         gperr nil
   )
   (gpuser)
   (setq sblip (getvar "blipmode"))
   (setq scmde (getvar "cmdecho"))
   (setq sang (getvar "snapang"))
   (setvar "blipmode" 0)
   (setvar "cmdecho" 0)
   (drawout)
   (gp_dialog)
   (drawtiles)
   (setvar "blipmode" sblip)
   (setvar "cmdecho" scmde)
   (setvar "snapang" sang)
   (setq *error* olderr)
   (princ)
)

; Print message once loaded.

(princ "\nDDGP.LSP Loaded. Type DDPATH to use.")
(princ)
```

Summary

See how quickly you've added a new command to AutoCAD. In many CAD systems you need to be an experienced programmer, master a larger body of knowledge, and have access to the CAD system's source code in order to do what you've just done. The open architecture of AutoCAD has put in your hands the power that most CAD system vendors reserve for themselves.

You can use this example as the first step in mastering AutoLISP. You might want to start by modifying and extending the PATH or DDPATH commands you've just completed. For an ambitious undertaking, make a new command that accepts a centre point and area and draws a square of the specified area filled with tiles.

You might also want to look through the functions you've just written in conjunction with the other chapters of this manual. Here we've given only brief descriptions of how the functions work and what you can make them do. There's more power buried in AutoLISP, and you can best use it by playing around and becoming familiar with the facilities it has to offer.

As you learn to use AutoLISP, you are moving to a new level of mastery in using AutoCAD. By using AutoLISP to automate your drafting and design tasks, you can improve your efficiency.

Appendix F
ASCII Codes

This appendix shows the standard ASCII codes. The octal value is useful in character or string constants, using the *\nnn* form. Your system might define additional codes in the extended, 256-character set (additional codes are greater than 127). Some systems also redefine some of the little-used ASCII codes such as 1–6 and 14–26. To see your system's characters with their codes in both decimal and octal form, run the following AutoLISP function, which prints to the screen and to a file called *ascii.txt*.

```
(defun C:ASCII ()
   (setq chk 2 code -1 ct 0)
   (textpage)
   (getstring "\nWriting to file ASCII.TXT. Press ENTER to continue.")
   (setq vvv (open "ascii.txt" "w"))
   (princ "\n \n CHARACTER DECIMAL OCTAL\n")
   (while (= chk 2)
      (setq  code (1+ code) ct (1+ ct)
             o1 (rtos (/ (/ code 8) 8) 2 0)
             o2 (rtos (rem (/ code 8) 8) 2 0)
             o3 (rtos (rem code 8) 2 0)
             oct (strcat o1 o2 o3))
      (princ (strcat "\n " (rtos code 2 0) "   " (chr code) "   " oct) vvv)
      (princ (strcat "\n\t" (chr code) "\t" (rtos code 2 0) "\t" oct))
      (if (= code 255)
         (setq chk 0)
         (if (= ct 20)
            (progn
               (setq xxx (getstring (strcat "\n \nPress 'X' to eXit or"
                  " any key to continue: ")))
               (if (= (strcase xxx) "X")
                  (setq chk 0)
                  (progn
                     (setq ct 0)
                     (princ "\n \n CHARACTER DECIMAL OCTAL\n")
                  )
               )
            )
         )
      )
   )
   (close vvv)
   (princ)
)
```

Dec.	Oct.	Hex.	Char.
0	000	00	NUL
1	001	01	SOH
2	002	02	STX
3	003	03	ETX
4	004	04	EOT
5	005	05	ENQ
6	006	06	ACK
7	007	07	BEL (bell)
8	010	08	BS (backspace)
9	011	09	HT
10	012	0A	LF (linefeed)
11	013	0B	VT
12	014	0C	FF
13	015	0D	CR (return)
14	016	0E	SO
15	017	0F	SI
16	020	10	DLE
17	021	11	DC1
18	022	12	DC2
19	023	13	DC3
20	024	14	DC4
21	025	15	NAK
22	026	16	SYN
23	027	17	ETB
24	030	18	CAN
25	031	19	EM
26	032	1A	SUB
27	033	1B	ESC (escape)
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US

Dec.	Oct.	Hex.	Char.
32	040	20	space
33	041	21	!
34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	’
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?

Dec.	Oct.	Hex.	Char.
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_

Dec.	Oct.	Hex.	Char.
96	140	60	‘
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL (delete)

Index