AutoCAD 2013

AutoLISP Developer's Guide

January 2012

Autodesk®

Contents

Introduction 1

Introduction

For years, AutoLISP® has set the standard for customizing AutoCAD® on Windows®. AutoCAD also supports AutoLISP, but does not support many of the Visual LISP functions or the Microsoft ActiveX® Automation interface. AutoCAD does not have an integrated development environment like AutoCAD on Windows does, so the creation and editing of LSP files must be done with text editor such as TextEdit.

AutoLISP

AutoLISP is a programming language designed for extending and customizing the functionality of AutoCAD. It is based on the LISP programming language, whose origins date back to the late 1950s. LISP was originally designed for use in Artificial Intelligence (AI) applications, and is still the basis for many AI applications.

AutoLISP was introduced as an application programming interface (API) in AutoCAD Release 2.1, in the mid-1980s. LISP was chosen as the initial AutoCAD API because it was uniquely suited for the unstructured design process of AutoCAD projects, which involved repeatedly trying different solutions to design problems.

Developing AutoLISP programs for AutoCAD is done by writing code in a text editor, then loading the code into AutoCAD and running it. Debugging your program is handled by adding statements to print the contents of variables at strategic points in your program. You must figure out where in your program to do this, and what variables you need to look at. If you discover you do not have enough information to determine the error, you must go back and change

the code by adding more debugging points. And finally, when you get the program to work correctly, you need to either comment out or remove the debugging code you added.

About Related AutoLISP Documents

In addition to the *AutoLISP Reference*, several other AutoCAD publications may be required by users building applications with AutoLISP:

- AutoCAD*Customization Guide* contains basic information on creating customized AutoCAD applications. For example, it includes information on creating customized user interface elements, linetypes, and hatch patterns. The *Customization Guide* is available through the AutoCAD and Help menu on the Mac OS menu bar.
- The *DXF Reference* describes drawing interchange format (DXF™) and the DXF group codes that identify attributes of AutoCAD objects.

  The *DXF Reference* is not included when you install AutoCAD. To obtain the manual, download the DXF Reference from www.autodesk.com.
- The *ObjectARX Reference* contains information on using ObjectARX® to develop customized AutoCAD applications. AutoCAD reactor functionality is implemented through ObjectARX. If you develop AutoLISP applications that implement reactor functions, you may want to refer to this manual.

  The *ObjectARX Reference* is not included when you install AutoCAD. To obtain the manual, download the ObjectARX SDK (Software Development Kit) from www.autodesk.com.

Using the AutoLISP Language

2

AutoLISP Basics

You can use number, string, and list-handling functions to customize AutoCAD.

This chapter introduces the basic concepts of the AutoLISP® programming language. It describes the core components and data types used in AutoLISP, and presents examples of simple number-, string-, output-, and list-handling functions.

AutoLISP code does not need to be compiled, so you can enter the code at a Command line and immediately see the results.

AutoLISP Expressions

An AutoLISP program consists of a series of expressions. AutoLISP expressions have the following form:

```
(function
arguments
)
```

Each expression begins with an open (left) parenthesis and consists of a function name and optional arguments to that function. Each argument can also be an expression. The expression ends with a right parenthesis. Every expression returns a value that can be used by a surrounding expression. The value of the last interpreted expression is returned to the calling expression.

For example, the following code example involves three functions:

```
(fun1 (fun2
arguments)(fun3
arguments)
)
```

If you enter this code at the AutoCAD Command prompt, the AutoCAD AutoLISP interpreter processes the code. The first function, **fun1**, has two arguments, and the other functions, **fun2** and **fun3**, each have one argument. The functions **fun2** and **fun3** are surrounded by function **fun1**, so their return values are passed to **fun1** as arguments. Function **fun1** evaluates the two arguments and returns the value to the window from which you entered the code.

The following example shows the use of the * (multiplication) function, which accepts one or more numbers as arguments:

```
(* 2 27)

54
```

Because this code example has no surrounding expression, AutoLISP returns the result to the window from which you entered the code.

Expressions nested within other expressions return their result to the surrounding expression. The following example uses the result from the + (addition) function as one of the arguments for the * (multiplication) function.

```
(* 2 (+ 5 10))

30
```

If you enter the incorrect number of close (right) parentheses, AutoLISP displays the following prompt:

```
(_>
```

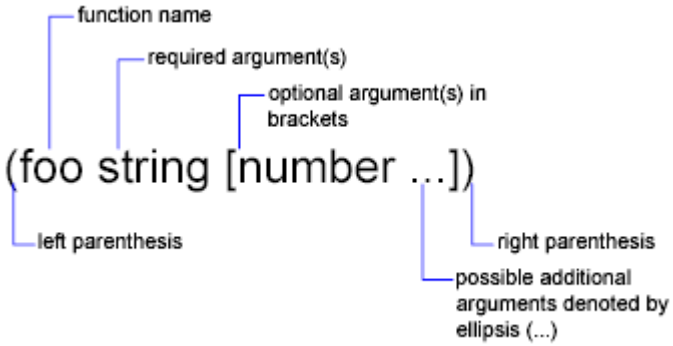
The number of open parentheses in this prompt indicates how many levels of open parentheses remain unclosed. If this prompt appears, you must enter the required number of close parentheses for the expression to be evaluated.

```
(* 2 (+ 5 10

((_>

) )

30
```

A common mistake is to omit the closing quotation mark (") in a text string, in which case the close parentheses are interpreted as part of the string and have no effect in resolving the open parentheses. To correct this condition, press Shift+Esc to cancel the function, then re-enter it correctly.

AutoLISP Function Syntax

In this guide, the following conventions describe the syntax for AutoLISP functions:



In this example, the **foo** function has one required argument, *string*, and one optional argument, *number*. Additional *number* arguments can be provided. Frequently, the name of the argument indicates the expected data type. The examples in the following table show both valid and invalid calls to the **foo** function.

Valid and invalid function call examples

Valid calls	Invalid calls
`(foo "catch")`	`(foo 44 13)`
`(foo "catch" 22)`	`(foo "fi" "foe" 44 13)`
`(foo "catch" 22 31)`	`(foo)`

AutoLISP Data Types

AutoLISP expressions are processed according to the order and data type of the code within the parentheses. Before you can fully utilize AutoLISP, you must understand the differences among the data types and how to use them.

Integers

Integers are whole numbers that do not contain a decimal point. AutoLISP integers are 32-bit signed numbers with values ranging from +2,147,483,647 to -2,147,483,648. (Note, however, that the **getint** function only accepts 16-bit numbers ranging from +32767 to -32678.) When you explicitly use an integer in an AutoLISP expression, that value is known as a constant. Numbers such as 2, -56, and 1,200,196 are valid AutoLISP integers.

If you enter a number that is greater than the maximum integer allowed (resulting in integer overflow), AutoLISP converts the integer to a real number. However, if you perform an arithmetic operation on two valid integers, and the result is greater than the maximum allowable integer, the resulting number will be invalid. The following examples illustrate how AutoLISP handles integer overflow.

The largest positive integer value retains its specified value:

2147483647

```
2147483647
```

If you enter an integer that is greater than the largest allowable value, AutoLISP returns the value as a real:

2147483648

```
2.14748e+009
```

An arithmetic operation involving two valid integers, but resulting in integer overflow, produces an invalid result:

(+ 2147483646 3)

```
-2147483647
```

In this example the result is clearly invalid, as the addition of two positive numbers results in a negative number. But note how the following operation produces a valid result:

(+ 2147483648 2)

```
2.14748e+009
```

In this instance, AutoLISP converts 2147483648 to a valid real before adding 2 to the number. The result is a valid real.

The largest negative integer value retains its specified value:

-2147483647

```
-2147483647
```

If you enter a negative integer larger than the greatest allowable negative value, AutoLISP returns the value as a real:

-2147483648

```
-2.14748e+009
```

The following operation concludes successfully, because AutoLISP first converts the overflow negative integer to a valid real:

(- -2147483648 1)

```
-2.14748e+009
```

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.

Reals can be expressed in scientific notation, which has an optional e or E followed by the exponent of the number (for example, 0.0000041 is the same as 4.1e-6). Numbers such as 3.1, 0.23, -56.123, and 21,000,000.0 are valid AutoLISP reals.

Strings

A string is a group of characters surrounded by quotation marks. Within quoted strings the backslash (\) character allows control characters (or escape codes) to be included. When you explicitly use a quoted string in an AutoLISP expression, that value is known as a literal string or a string constant.

Examples of valid strings are “string 1” and “\nEnter first point:”.

Lists

An AutoLISP list is a group of related values separated by spaces and enclosed in parentheses. Lists provide an efficient method of storing numerous related values. AutoCAD expresses 3D points as a list of three real numbers.

Examples of lists are (1.0 1.0 0.0), (“this” “that” “the other”), and (1 “ONE”).

Selection Sets

Selection sets are groups of one or more objects (entities). You can interactively add objects to, or remove objects from, selection sets with AutoLISP routines.

The following example uses the **ssget** function to return a selection set containing all the objects in a drawing.

```
(ssget "X")

<Selection set: 1>
```

Entity Names

An entity name is a numeric label assigned to objects in a drawing. It is actually a pointer into a file maintained by AutoCAD, and can be used to find the object's database record and its vectors (if they are displayed). This label can be referenced by AutoLISP functions to allow selection of objects for processing in various ways. Internally, AutoCAD refers to objects as entities.

The following example uses the **entlast** function to get the name of the last object entered into the drawing.

(entlast)

```
<Entity name: 27f0540>
```

Entity names assigned to objects in a drawing are only in effect during the current editing session. The next time you open the drawing, AutoCAD assigns new entity names to the objects. You can use an object's handle to refer to it from one editing session to another; see Entity Handles and Their Uses (page 87) for information on using handles.

File Descriptors

A file descriptor is a pointer to a file opened by the AutoLISP **open** function. The **open** function returns this pointer as an alphanumeric label. You supply the file descriptor as an argument to other AutoLISP functions that read or write to the file.

The following example opens the *myinfo.dat* file for reading. The **open** function returns the file descriptor:

(setq file1 (open "/myinfo.dat" "r"))

```
#<file "/myinfo.dat">
```

In this example, the file descriptor is stored in the `file1`variable.

Files remain open until you explicitly close them in your AutoLISP program. The **close** function closes a file. The following code closes the file whose file descriptor is stored in the `file1` variable:

(close file1)

```
nil
```

Symbols and Variables

AutoLISP uses symbols to refer to data. Symbol names are not case sensitive and may consist of any sequence of alphanumeric and notation characters, except the following:

Characters restricted from symbol names	
(	(Open Parenthesis)
)	(Close Parenthesis)
.	(Period)
'	(Apostrophe)
"	(Quote Symbol)
;	(Semicolon)

A symbol name cannot consist only of numeric characters.

Technically, AutoLISP applications consist of either symbols or constant values, such as strings, reals, and integers. For the sake of clarity, this guide uses the term *symbol* to refer to a symbol name that stores static data, such as built-in and user-defined functions. The term *variable* is used to refer to a symbol name that stores program data. The following example uses the **setq** function to assign the string value `"this is a string"` to the `str1` variable:

```
(setq str1 "this is a string")

"this is a string"
```

Help yourself and others who need to read your code. Choose meaningful names for your program symbols and variables.

AutoLISP Program Files

Although you can enter AutoLISP code at the AutoCAD Command prompt, testing and debugging a series of instructions are considerably easier when you save AutoLISP code in a file rather than re-entering it each time you make a refinement. AutoLISP source code is usually stored in ASCII text files with an *.lsp* extension. However, you can load AutoLISP code from any ASCII text file.

Formatting AutoLISP Code

The extensive use of parentheses in AutoLISP code can make it difficult to read. The traditional technique for combatting this confusion is indentation. The more deeply nested a line of code is, the farther to the right you position the line.

Spaces in AutoLISP Code

In AutoLISP, multiple spaces between variable names, constants, and function names are equivalent to a single space. The end of a line is also treated as a single space.

The following two expressions produce the same result:

```
(setq test1 123 test2 456)
(setq
    test1 123
    test2 456
)
```

Comments in AutoLISP Program Files

It is good practice to include comments in AutoLISP program files. Comments are useful to both the programmer and future users who may need to revise a program to suit their needs. Use comments to do 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

Comments begin with one or more semicolons (;) and continue through the end of the line.

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

Any text within ;| ... |; is ignored. Therefore, comments can be included within a line of code or extend for multiple lines. This type of comment is known as an in-line comment.

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

The following example 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.")
```

It is recommended that you use comments liberally when writing AutoLISP programs.

AutoLISP Variables

An AutoLISP variable assumes the data type of the value assigned to it. Until they are assigned new values, variables retain their original values. You use the AutoLISP **setq** function to assign values to variables.

```
(setq
  variable_name1 value1 [variable_name2 value2 ...]
)
```

The **setq** function assigns the specified value to the variable name given. It returns the value as its function result.

(setq val 3 abc 3.875)

```
3.875
```

(setq layr "EXTERIOR-WALLS")

```
"EXTERIOR-WALLS"
```

Displaying the Value of a Variable

To display the value of a variable from the AutoCAD Command prompt, you must precede the variable name with an exclamation point (!). For example:

```
!abc

3.875
```

Nil Variables

An AutoLISP variable that has not been assigned a value is said to be `nil`. This is different from blank, which is considered a character string, and different from 0, which is a number. So, in addition to checking a variable for its current value, you can test to determine if the variable has been assigned a value.

Each variable consumes a small amount of memory, so it is good programming practice to reuse variable names or set variables to `nil` when their values are no longer needed. Setting a variable to `nil` releases the memory used to store that variable's value. If you no longer need the `val` variable, you can release its value from memory with the following expression:

```
(setq val nil)

nil
```

Another efficient programming practice is to use local variables whenever possible. See Local Variables in Functions (page 34) on this topic.

Predefined Variables

The following predefined variables are commonly used in AutoLISP applications:

`PAUSE` Defined as a string consisting of a double backslash (\\) character. This variable is used with the **command** function to pause for user input.

`PI` Defined as the constant p (pi). It evaluates to approximately 3.14159.

`T` Defined as the constant `T`. This is used as a non-`nil` value.

NOTE You can change the value of these variables with the **setq** function. However, other applications might rely on their values being consistent; therefore, it is recommended that you do not modify these variables.

Number Handling

AutoLISP provides functions for working with integers and real numbers. In addition to performing complex mathematical computations in applications, you can use the number-handling functions to help you in your daily use of AutoCAD. If you are drawing a steel connection detail that uses a 2.5" bolt that is 0.5" in diameter, how many threads are there if the bolt has 13 threads per inch?

```
(* 2.5 13)

32.5
```

The arithmetic functions that have a `number` argument (as opposed to `num` or `angle`, for example) return different values if you provide integers or reals as arguments. If all arguments are integers, the value returned is an integer. However, if one or all the arguments are reals, the value returned is a real. To ensure your application passes real values, be certain at least one argument is a real.

```
(/ 12 5)

2

(/ 12.0 5)

2.4
```

A complete list of number-handling functions is in AutoLISP Function Synopsis, (page 119) under the heading Arithmetic Functions. (page 122) These functions are described in the *AutoLISP Reference*.

String Handling

AutoLISP provides functions for working with string values. For example, the **strcase** function returns the conversion of all alphabetic characters in a string to uppercase or lowercase. It accepts two arguments: a string and an optional

argument that specifies the case in which the characters are returned. If the optional second argument is omitted, it evaluates to `nil` and **strcase** returns the characters converted to uppercase.

(strcase "This is a TEST.")

```
"THIS IS A TEST."
```

If you provide a second argument of `T`, the characters are returned as lowercase. AutoLISP provides the predefined variable `T` to use in similar situations where a non-`nil` value is used as a type of true/false toggle.

(strcase "This is a TEST." T)

```
"this is a test."
```

The **strcat** function combines multiple strings into a single string value. This is useful for placing a variable string within a constant string. The following code sets a variable to a string value and then uses **strcat** to insert that string into the middle of another string.

(setq str "BIG") (setq bigstr (strcat "This is a " str " test."))

```
"This is a BIG test."
```

If the variable `bigstr` is set to the preceding string value, you can use the **strlen** function to find out the number of characters (including spaces) in that string.

(strlen bigstr)

```
19
```

The **substr** function returns a substring of a string. It has two required arguments and one optional argument. The first required argument is the string. The second argument is a positive integer that specifies the first character of the string you want to include in the substring. If the third argument is provided, it specifies the number of characters to include in the substring. If the third argument is not provided, **substr** returns all characters including and following the specified start character.

As an example, you can use the **substr** function to strip off the three-letter extension from a file name (note that you can actually use the **vl-filename-base** function to do this). First, set a variable to a file name.

(setq filnam "bigfile.txt")

```
"bigfile.txt"
```

You need to get a string that contains all characters except the last four (the period and the three-letter extension). Use **strlen** to get the length of the string and subtract 4 from that value. Then use **substr** to specify the first character of the substring and its length.

(setq newlen (- (strlen filnam) 4))

```
7
```

(substr filnam 1 newlen)

```
"bigfile"
```

If your application has no need for the value of `newlen`, you can combine these two lines of code into one.

(substr filnam 1 (- (strlen filnam) 4))

```
"bigfile"
```

Additional string-handling functions are listed in AutoLISP Function Synopsis, (page 119) under the heading String-Handling Functions. (page 131) These functions are described in the *AutoLISP Reference*.

AutoLISP also provides a number of functions that convert string values into numeric values and numeric values into string values. These functions are discussed in Conversions (page 61).

Basic Output Functions

AutoLISP includes functions for controlling the AutoCAD display, including both text and graphics windows. The major text display functions are:

- **prin1**
- **princ**
- **print**
- **prompt**

These functions are discussed in the following sections. The remaining display functions are covered in Using AutoLISP to Communicate with AutoCAD (page 42), beginning with the Display Control (page 47) topic.

Displaying Messages

The **princ**, **prin1**, and **print** functions all display an expression (not necessarily a string) in the AutoCAD Command window. Optionally, these functions can send output to a file. The differences are as follows:

- **princ** displays strings without the enclosing quotation marks.
- **prin1** displays strings enclosed in quotation marks.
- **print** displays strings enclosed in quotation marks but places a blank line before the expression and a space afterward.

The following examples demonstrate the differences between the four basic output functions and how they handle the same string of text. See Control Characters in Strings (page 18) for an explanation of the control characters used in the example.

```
(setq str "The \"allowable\" tolerance is \261 \274\"")
(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 1/4\""

(prin1 str)
prints
```
"The \"allowable\" tolerance is 1/4"
```
"and returns "The
\"allowable\" tolerance is  1/4\""

(print str)
prints<blank line>
```
"The \"allowable\" tolerance is 1/4"
```
"<space>and returns  "The \"allowable\" tolerance is
1/4\""
```

Note that the **write-char** and **write-line** functions can also display output to a Command window. Refer to the *AutoLISP Reference* for information on these functions.

Exiting Quietly

If you invoke the `princ` function without passing an expression to it, it displays nothing and has no value to return. So if you write an AutoLISP expression that ends with a call to **princ** without any arguments, the ending `nil` is suppressed (because it has nothing to return). This practice is called exiting quietly.

Control Characters in Strings

Within quoted strings, the backslash (\) character allows control characters (or escape codes) to be included. The following table shows the currently recognized control characters:

AutoLISP control characters

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

The **prompt** and **princ** functions expand the control characters in a string and display the expanded string in the AutoCAD Command window.

If you need to use the backslash character (\) or quotation mark (") within a quoted string, it must be preceded by the backslash character (\). For example, if you enter

(princ "The \"filename\" is: /ACAD/TEST.TXT.")

the following text is displayed in the AutoCAD Command window:

```
The "filename" is: /ACAD/TEST.TXT
```

You will also see this output in the VLISP Console window, along with the return value from the **princ** function (which is your original input, with the unexpanded control characters).

To force a line break at a specific location in a string, use the newline character (`\n`).

(prompt "An example of the \nnewline character. ")

```
An example of the
newline character.
```

You can also use the **terpri** function to cause a line break.

The return character (`\r`) returns to the beginning of the current line. This is useful for displaying incremental information (for example, a counter showing the number of objects processed during a loop).

The Tab character (\t) can be used in strings to indent or to provide alignment with other tabbed text strings. In this example, note the use of the **princ** function to suppress the ending `nil`.

(prompt "\nName\tOffice\n- - - - -**\t**- - - - -
(_>
\nSue\t101\nJoe\t102\nSam\t103\n") (princ)

Name	Office
- - - - -	- - - - -
Sue	101
Joe	102
Sam	103

Wild-Card Matching

The **wcmatch** function enables applications to compare a string to a wild-card pattern. You can use this facility when you build a selection set (in conjunction with **ssget**) and when you retrieve extended entity data by application name (in conjunction with **entget**).

The **wcmatch** function compares a single string to a pattern. The function 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; an asterisk (*) matches a sequence of characters; and, certain other special characters have special meanings within the pattern. When you use the `*` character at the beginning and end of the search pattern, you can locate the desired portion anywhere in the string.

In the following examples, a string variable called `matchme` has been declared and initialized:

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

"this is a string - test1 test2 the end"
```

The following code checks whether or not `matchme` begins with the four characters `"this"`:

```
(wcmatch matchme "this*")

T
```

The following code illustrates the use of brackets in the pattern. In this case, **wcmatch** returns `T` if `matchme` contains `"test4"`, `"test5"`, `"test6" (4-6)`, or `"test9"` (note the use of the `*` character):

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

nil
```

In this case, **wcmatch** returns `nil` because `matchme` does not contain any of the strings indicated by the pattern.

However,

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

T
```

returns `T` because the string contains `"test1"`.

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

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

T
```

Equality and Conditional

AutoLISP includes functions that provide equality verification as well as conditional branching and looping. The equality and conditional functions are listed in AutoLISP Function Synopsis, (page 119) under the heading Equality and Conditional Functions. (page 125) These functions are described in the *AutoLISP Reference*.

When writing code that checks string and symbol table names, keep in mind that AutoLISP automatically converts symbol table names to upper case in some instances. When testing symbol names for equality, you need to make the comparison insensitive to the case of the names. Use the **strcase** function to convert strings to the same case before testing them for equality.

List Handling

AutoLISP provides functions for working with lists. This section provides examples of the **append**, **assoc**, **car**, **cons**, **list**, **nth**, and **subst** functions. A summary of all list-handling functions is in AutoLISP Function Synopsis, (page 119) under the heading List Manipulation Functions. (page 128) Each list-handling function is described in the *AutoLISP Reference*.

Lists provide an efficient and powerful method of storing numerous related values. After all, LISP is so-named because it is the LISt Processing language. Once you understand the power of lists, you'll find that you can create more powerful and flexible applications.

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.

The **list** function provides a simple method of grouping related items. These items do not need to be of similar data types. The following code groups three related items as a list:

```
(setq lst1 (list 1.0 "One" 1))

(1.0 "One" 1)
```

You can retrieve a specific item from the list in the `lst1` variable with the **nth** function. This function accepts two arguments. The first argument is an integer that specifies which item to return. A 0 specifies the first item in a list, 1 specifies the second item, and so on. The second argument is the list itself. The following code returns the second item in `lst1`.

```
(nth 1 lst1)

"One"
```

The **cdr** function returns all elements, except the first, from a list. For example:

```
(cdr lst1)

("One" 1)
```

The **car** function provides another way to extract items from a list. For more examples using **car** and **cdr**, and combinations of the two, see Point Lists (page 23).

Three functions let you modify an existing list. The **append** function returns a list with new items added to the end of it, and the **cons** function returns a list with new items added to the beginning of the list. The **subst** function returns a list with a new item substituted for every occurrence of an old item. These functions do not modify the original list; they return a modified list. To modify the original list, you must explicitly replace the old list with the new list.

The **append** function takes any number of lists and runs them together as one list. Therefore, all arguments to this function must be lists. The following code adds another `"One"` to the list `lst1`. Note the use of the **quote** (or ') function as an easy way to make the string `"One"` into a list.

(setq lst2 (append lst1 '("One")))

```
(1.0 "One" 1 "One")
```

The **cons** function combines a single element with a list. You can add another string `"One"` to the beginning of this new list, `lst2`, with the **cons** function.

(setq lst3 (cons "One" lst2))

```
("One" 1.0 "One" 1 "One")
```

You can substitute all occurrences of an item in a list with a new item with the **subst** function. The following code replaces all strings `"One"` with the string `"one"`.

(setq lst4 (subst "one" "One" lst3))

```
("one" 1.0 "one" 1 "one")
```

Point Lists

AutoLISP observes the following conventions for handling graphics coordinates. Points are expressed as *lists* of two or three numbers surrounded by parentheses.

2D points Expressed as lists of two real numbers (*X* and *Y*, respectively), as in

```
(3.4 7.52)
```

3D points Expressed as lists of three real numbers (*X*, *Y*, and *Z*, respectively), as in

```
(3.4 7.52 1.0)
```

You can use the **list** function to form point lists, as shown in the following examples:

(list 3.875 1.23)

```
(3.875 1.23)
```

(list 88.0 14.77 3.14)

```
(88.0 14.77 3.14)
```

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

(setq pt1 (list 3.875 1.23))

```
(3.875 1.23)
```

(setq pt2 (list 88.0 14.77 3.14))

```
(88.0 14.77 3.14)
```

(setq abc 3.45)

```
3.45
```

(setq pt3 (list abc 1.23))

```
(3.45 1.23)
```

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

If all members of a list are constant values, you can use the **quote** function to explicitly define the list, rather than the **list** function. The **quote** function returns an expression without evaluation, as follows:

(setq pt1 (quote (4.5 7.5)))

```
(4.5 7.5)
```

The single quotation mark (') can be used as shorthand for the **quote** function. The following code produces the same result as the preceding code.

(setq pt1 '(4.5 7.5))

```
(4.5 7.5)
```

You can refer to *X*, *Y*, and *Z* components of a point individually, using three additional built-in functions called **car**, **cadr**, and **caddr**. The following examples show how to extract the *X*, *Y*, and *Z* coordinates from a 3D point list. The `pt` variable is set to the point `(1.5 3.2 2.0)`:

(setq pt '(1.5 3.2 2.0))

```
(1.5 3.2 2.0)
```

The **car** function returns the first member of a list. In this example it returns the *X* value of point `pt` to the `x_val` variable.

```
(setq x_val (car pt))

1.5
```

The **cadr** function returns the second member of a list. In this example it returns the *Y* value of the `pt` point to the `y_val` variable.

```
(setq y_val (cadr pt))

3.2
```

The **caddr** function returns the third member of a list. In this example it returns the *Z* value of point `pt` to the variable `z_val`.

```
(setq z_val (caddr pt))

2.0
```

You can use the following code to define the lower-left and upper-right (`pt1` and `pt2`) corners of a rectangle, as follows:

```
(setq pt1 '(1.0 2.0) pt2 ' (3.0 4.0))

(3.0 4.0)
```

You can use the **car** and **cadr** functions to set the `pt3` variable 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)))

(1.0 4.0)
```

The preceding expression sets `pt3` equal to point `(1.0,4.0)`.

AutoLISP supports concatenations of **car** and **cdr** up to four levels deep. The following are valid functions:

caaaar	**cadaar**	**cdaaar**	**cddaar**
caaadr	**cadadr**	**cdaadr**	**cddadr**

caaar	**cadar**	**cdaar**	**cddar**
caadar	**caddar**	**cdadar**	**cdddar**
caaddr	**cadddr**	**cdaddr**	**cddddr**
caadr	**caddr**	**cdadr**	**cdddr**
caar	**cadr**	**cdar**	**cddr**

These concatenations are the equivalent of nested calls to **car** and **cdr**. Each *a* represents a call to **car**, and each *d* represents a call to **cdr**. For example:

```
(caar x)
is equivalent to (car (car x))

(cdar x)
is equivalent to (cdr (car x))

(cadar x)
is equivalent to (car (cdr (car x)))

(cadr x)
is equivalent to (car (cdr x))

(cddr x)
is equivalent to (cdr (cdr x))

(caddr x)
is equivalent to (car (cdr (cdr x)))
```

Dotted Pairs

Another way AutoLISP uses lists to organize data is with a special type of list called a dotted pair. This list must always contain two members. When representing a dotted pair, AutoLISP separates the members of the list with a period (**.**). Most list-handling functions will not accept a dotted pair as an argument, so you should be sure you are passing the right kind of list to a function.

Dotted pairs are an example of an "improper list." An improper list is one in which the last `cdr` is not `nil`. In addition to adding an item to the beginning of a list, the **cons** function can create a dotted pair. If the second argument to the **cons** function is anything other than another list or `nil`, it creates a dotted pair.

(setq sublist (cons 'lyr "WALLS"))

```
(LYR . "WALLS")
```

The **car**, **cdr**, and **assoc** functions are useful for handling dotted pairs. The following code creates an association list, which is a list of lists, and is the method AutoLISP uses to maintain entity definition data. (Entity definition data is discussed in Using AutoLISP to Manipulate AutoCAD Objects. (page 74)) The following code creates an association list of dotted pairs:

(setq wallinfo (list sublist(cons 'len 240.0) (cons 'hgt 96.0)))

```
( (LYR . "WALLS")  (LEN . 240.0)  (HGT . 96.0) )
```

The **assoc** function returns a specified list from within an association list regardless of the specified list's location within the association list. The **assoc** function searches for a specified key element in the lists, as follows:

(assoc 'len wallinfo)

```
(LEN  .  240.0)
```

(cdr (assoc 'lyr wallinfo))

```
"WALLS"
```

(nth 1 wallinfo)

```
(LEN  .  240.0)
```

(car (nth 1 wallinfo))

```
LEN
```

Symbol and Function Handling

AutoLISP provides a number of functions for handling symbols and variables. The symbol-handling functions are listed in AutoLISP Function Synopsis, (page 119) under the heading Symbol-Handling Functions (page 133) Each symbol-handling function is described in the *AutoLISP Reference*.

AutoLISP provides functions for handling one or more groups of functions. This section provides examples of the **defun** function. The remaining function-handling functions are listed in AutoLISP Function Synopsis, (page 119) under the heading Symbol-Handling Functions (page 133) The functions are described in the *AutoLISP Reference*.

Using defun to Define a Function

With AutoLISP, you can define your own functions. Once defined, these functions can be used at the AutoCAD Command prompt, the Visual LISP Console prompt, or within other AutoLISP expressions, just as you use the standard functions. You can also create your own AutoCAD commands, because commands are just a special type of function.

The **defun** function combines a group of expressions into a function or command. This function requires at least three arguments, the first of which is the name of the function (symbol name) to define. The second argument is the argument list (a list of arguments and local variables used by the function). The argument list can be `nil` or an empty list `()`. Argument lists are discussed in greater detail in Functions with Arguments (page 36). If local variables are provided, they are separated from the arguments by a slash (/). Local variables are discussed in Local Variables in Functions (page 34). Following these arguments are the expressions that make up the function; there must be at least one expression in a function definition.

```
(defun symbol_name (args / local_variables) expressions)
```

The following code defines a simple function that accepts no arguments and displays “bye” in the AutoCAD Command window. Note that the argument list is defined as an empty list (**()**):

```
(defun DONE ( ) (prompt "\nbye! "))

DONE
```

Now that the **DONE** function is defined, you can use it as you would any other function. For example, the following code prints a message, then says “bye” in the AutoCAD Command window:

(prompt "The value is 127.") (DONE) (princ)

```
The value is 127 bye!
```

Note how the previous example invokes the **princ** function without any arguments. This suppresses an ending `nil` and achieves a quiet exit.

Functions that accept no arguments may seem useless. However, you might use this type of function to query the state of certain system variables or conditions and to return a value that indicates those values.

AutoCAD can automatically load your functions each time you start a new AutoCAD session or open a new AutoCAD drawing file.

Any code in an AutoLISP program file that is not part of a **defun** statement is executed when that file is loaded. You can use this to set up certain parameters or to perform any other initialization procedures in addition to displaying textual information, such as how to invoke the loaded function.

Compatibility of defun with Previous Versions of AutoCAD

The internal implementation of **defun** changed in AutoCAD 2000. This change will be transparent to the great majority of AutoLISP users upgrading from earlier versions of AutoCAD. The change only affects AutoLISP code that manipulated **defun** definitions as a list structure, such as by appending one function to another, as in the following code:

```
(append s::startup (cdr mystartup))
```

For situations like this, you can use **defun-q** to define your functions. An attempt to use a **defun** function as a list results in an error. The following example illustrates the error:

(defun foo (x) 4)

```
foo
```

(append foo '(3 4))

```
; error: Invalid attempt to access a compiled function
definition.
You may want to define it using defun-q: #<SUBR @024bda3c
 FOO>
```

The error message alerts you to the possibility of using **defun-q** instead of **defun**.

The **defun-q** function is provided strictly for backward compatibility with previous versions of AutoLISP and should not be used for other purposes. For more information on using **defun-q**, and the related **defun-q-list-set** and **defun-q-list-ref** functions, see the *AutoLISP Reference*.

C:XXX Functions

If an AutoLISP function is defined with a name of the form C:***xxx***, it can be issued at the AutoCAD Command prompt in the same manner as a built-in AutoCAD command. You can use this feature to add new commands to AutoCAD or to redefine existing commands.

To use functions as AutoCAD commands, be sure they adhere to the following rules:

- The function name must use the form **C:*XXX*** (upper- or lowercase characters). The **C:** portion of the name must always be present; the ***XXX*** portion is a command name of your choice. **C:*XXX*** functions can be used to override built-in AutoCAD commands. (See Redefining AutoCAD Commands (page 32).)
- The function must be defined with no arguments. However, local variables are permitted and it is a good programming practice to use them.

A function defined in this manner can be issued transparently from within any prompt of any built-in AutoCAD command, provided the function issued transparently does not call the **command** function. (This is the AutoLISP function you use to issue AutoCAD commands; see the entry on command in the *AutoLISP Reference*.) When issuing a **C:*XXX*** defined command transparently, you must precede the ***XXX*** portion with a single quotation mark (').

You can issue a built-in command transparently while a **C:*XXX*** command is active by preceding it with a single quotation mark ('), as you would with all

commands that are issued transparently. However, you cannot issue a **C:*XXX***command transparently while a **C:*XXX*** command is active.

NOTE When calling a function defined as a command from the code of another AutoLISP function, you must use the whole name, including the parentheses; for example, **(C:HELLO)**. You also must use the whole name and the parentheses when you invoke the function from the VLISP Console prompt.

Adding Commands

Using the **C:*XXX*** feature, you can define a command that displays a simple message.

```
(defun C:HELLO () (princ "Hello world. \n") (princ))

C:HELLO
```

HELLO is now defined as a command, in addition to being an AutoLISP function. This means you can issue the command from the AutoCAD Command prompt.

```
Command: hello
Hello world.
```

This new command can be issued transparently because it does not call the **command** function itself. At the AutoCAD Command prompt, you could do the following:

```
Command: line
From point: 'hello
Hello world.
From point:
```

If you follow your function definition with a call to the **setfunhelp** function, you can associate a Help file and topic with a user-defined command. When help is requested during execution of the user-defined command, the topic specified by **setfunhelp** displays. See the *AutoLISP Reference* for more information on using **setfunhelp**.

You cannot usually use an AutoLISP statement to respond to prompts from an AutoLISP-implemented command. However, if your AutoLISP routine makes use of the **initget** function, you can use arbitrary keyboard input with certain functions. This allows an AutoLISP-implemented command to accept an AutoLISP statement as a response. Also, the values returned by a DIESEL expression can perform some evaluation of the current drawing and return

these values to AutoLISP. See Keyword Options (page 53) for more information on using **initget**, and refer to the AutoCAD*Customization Guide* for information on the DIESEL string expression language.

Redefining AutoCAD Commands

Using AutoLISP, external commands, and the alias feature, you can define your own AutoCAD commands. You can use the UNDEFINE command to redefine a built-in AutoCAD command with a user-defined command of the same name. To restore the built-in definition of a command, use the REDEFINE command. The UNDEFINE command is in effect for the current editing session only.

You can always activate an undefined command by specifying its true name, which is the command name prefixed by a period. For example, if you undefine QUIT, you can still access the command by entering **.quit** at the AutoCAD Command prompt. This is also the syntax that should be used within the AutoLISP **command** function.

Consider the following example. Whenever you use the LINE command, you want AutoCAD to remind you about using the PLINE command. You can define the AutoLISP function **C:LINE** to substitute for the normalLINEcommand as follows:

```
(defun C:LINE ( )
(_>

(princ "Shouldn't you be using PLINE?\n")
(_>

(command ".LINE") (princ))
C:LINE
```

In this example, the function **C:LINE** is designed to issue its message and then to execute the normal LINE command (using its true name, .LINE). Before AutoCAD will use your new definition for the LINE command, you must undefine the built-in LINE command. Enter the following to undefine the built-in LINE command:

```
(command "undefine" "line")
```

Now, if you enter **line** at the AutoCAD Command prompt, AutoCAD uses the **C:LINE** AutoLISP function:

```
Command: line
Shouldn't you be using PLINE?
.LINE Specify first point: Specify first point:
```

The previous code example assumes the CMDECHO system variable is set to 1 (On). If CMDECHO is set to 0 (Off), AutoCAD does not echo prompts during a **command** function call. The following code uses the CMDECHO system variable to prevent the LINE command prompt from repeating:

```
(defun C:LINE ( / cmdsave )
(_>

(setq cmdsave (getvar "cmdecho"))
(_>

(setvar "cmdecho" 0)
(_>

(princ "Shouldn't you be using PLINE?\n")
(_>

(command ".LINE")
(_>

(setvar "cmdecho" cmdsave)
(_>

(princ))
C:LINE
```

Now if you enter **line** at the AutoCAD Command prompt, the following text is displayed:

```
Shouldn't you be using PLINE?
Specify first point:
```

You can use this feature in a drawing management system, for example. You can redefine the NEW, OPEN, and QUIT commands to write billing information to a log file before you terminate the editing session.

It is recommended that you protect your menus, scripts, and AutoLISP programs by using the period-prefixed forms of all commands. This ensures that your applications use the built-in command definitions rather than a redefined command.

See the Overview of File Organization topic in the AutoCAD*Customization Guide* for a description of the steps AutoCAD takes to evaluate command names.

Local Variables in Functions

AutoLISP provides a method for defining a list of symbols (variables) that are available only to your function. These are known as local variables.

Local Variables versus Global Variables

The use of local variables ensures that the variables in your functions are unaffected by the surrounding application and that your variables do not remain available after the calling function has completed its task.

Many user-defined functions are used as utility functions within larger applications. User-defined functions also typically contain a number of variables whose values and use are specific to that function.

The danger in using global variables, instead of local variables, is you may inadvertently modify them outside of the function they were declared in and intended for. This can lead to unpredictable behavior, and it can be very difficult to identify the source of this type of problem.

Another advantage of using local variables is that AutoCAD can recycle the memory space used by these variables, whereas global variables keep accumulating within AutoCAD memory space.

There are some legitimate uses for global variables, but these should be kept to a minimum. It is also a good practice to indicate that you intend a variable to be global. A common way of doing this is to add an opening and closing asterisk to the variable name, for example, `*default-layer*`.

Example Using Local Variables

The following example shows the use of local variables in a user-defined function (be certain there is at least one space between the slash and the local variables).

```
(defun LOCAL ( / aaa bbb)
  (_>
```

```
(setq aaa "A" bbb "B")
(_>

(princ (strcat "\naaa has the value " aaa ))
(_>

(princ (strcat "\nbbb has the value " bbb))
(_>

(princ))
LOCAL
```

Before you test the new function, assign variables `aaa` and `bbb` to values other than those used in the **LOCAL** function.

```
(setq aaa 1 bbb 2)

2
```

You can verify that the variables `aaa` and `bbb` are actually set to those values.

```
aaa

1

bbb

2
```

Now test the **LOCAL** function.

```
(local)

aaa has the value A
bbb has the value B
```

You will notice the function used the values for `aaa` and `bbb` that are local to the function. You can verify that the current values for `aaa` and `bbb` are still set to their nonlocal values.

```
aaa

1
```

```
bbb

2
```

In addition to ensuring that variables are local to a particular function, this technique also ensures the memory used for those variables is available for other functions.

Functions with Arguments

With AutoLISP, you can define functions that accept arguments. Unlike many of the standard AutoLISP functions, user-defined functions cannot have optional arguments. When you call a user-defined function that accepts arguments, you must provide values for all the arguments.

The symbols to use as arguments are defined in the argument list before the local variables. Arguments are treated as a special type of local variable; argument variables are not available outside the function. You cannot define a function with multiple arguments of the same name.

The following code defines a function that accepts two string arguments, combines them with another string, and returns the resulting string.

```
(defun ARGTEST ( arg1 arg2 / ccc )
(_>

(setq ccc "Constant string")
(_>

(strcat ccc ", " arg1 ", " arg2))

ARGTEST
```

The **ARGTEST** function returns the desired value because AutoLISP always returns the results of the last expression it evaluates. The last line in **ARGTEST** uses **strcat** to concatenate the strings, and the resulting value is returned. This is one example where you should not use the **princ** function to suppress the return value from your program.

This type of function can be used a number of times within an application to combine two variable strings with one constant string in a specific order. Because it returns a value, you can save the value to a variable for use later in the application.

(setq newstr (ARGTEST "String 1" "String 2"))

```
"Constant string, String 1, String 2"
```

The `newstr` variable is now set to the value of the three strings combined.

Note that the `ccc` variable was defined locally within the **ARGTEST** function. Once the function runs to completion, AutoLISP recycles the variable, recapturing the memory allocated to it. To prove this, check from the VLISP Console window to see if there is still a value assigned to `ccc`.

ccc

```
nil
```

Special Forms

Certain AutoLISP functions are considered special forms because they evaluate arguments in a different manner than most AutoLISP function calls. A typical function evaluates all arguments passed to it before acting on those arguments. Special forms either do not evaluate all their arguments, or only evaluate some arguments under certain conditions.

The following AutoLISP functions are considered special forms:

- **AND**
- **COMMAND**
- **COND**
- **DEFUN**
- **DEFUN-Q**
- **FOREACH**
- **FUNCTION**
- **IF**
- **LAMBDA**
- **OR**
- **PROGN**
- **QUOTE**
- **REPEAT**

- **SETQ**
- **TRACE**
- **UNTRACE**
- **VLAX-FOR**
- **WHILE**

You can read about each of these functions in the *AutoLISP Reference*.

Error Handling in AutoLISP

The AutoLISP language provides several functions for error handling. You can use these functions to do the following:

- Provide information to users when an error occurs during the execution of a program.
- Restore the AutoCAD environment to a known state.
- Intercept errors and continue program execution.

The complete list of error-handling functions is in AutoLISP Function Synopsis, (page 119) under the heading Error-Handling Functions. (page 126) Each error-handling function is described in the *AutoLISP Reference*.

If your program contains more than one error in the same expression, you cannot depend on the order in which AutoLISP detects the errors. For example, the **inters** function requires several arguments, each of which must be either a 2D or 3D point list. A call to **inters** like the following:

```
(inters 'a)
```

is an error on two counts: too few arguments and invalid argument type. You will receive either of the following error messages:

```
; *** ERROR: too few arguments
; *** ERROR: bad argument type: 2D/3D point
```

Your program should be designed to handle either error.

Note also that in AutoCAD, AutoLISP evaluates all arguments before checking the argument types. In previous releases of AutoCAD, AutoLISP evaluated and checked the type of each argument sequentially. To see the difference, look at the following code examples:

```
(defun foo ()
  (print "Evaluating foo")
  '(1 2))

(defun bar ()
  (print "Evaluating bar")
  'b)

(defun baz ()
  (print "Evaluating baz")
  'c)
```

Observe how an expression using the **inters** function is evaluated in AutoCAD:

```
Command: (inters (foo) (bar) (baz))
"Evaluating foo"
"Evaluating bar"
"Evaluating baz"
; *** ERROR: too few arguments
```

Each argument was evaluated successfully before AutoLISP passed the results to **inters** and discovered that too few arguments were specified.

In AutoCAD R14 or earlier, the same expression evaluated as follows:

```
Command: (inters (foo) (bar) (baz))
"Evaluating foo"
"Evaluating bar" error: bad argument type
```

AutoLISP evaluated **(foo)**, then passed the result to **inters**. Since the result was a valid 2D point list, AutoLISP proceeds to evaluate **(bar)**, where it determines that the evaluated result is a string, an invalid argument type for **inters**.

Using the *error* Function

Proper use of the ***error*** function can ensure that AutoCAD returns to a particular state after an error occurs. Through this user-definable function you can assess the error condition and return an appropriate message to the user. If AutoCAD encounters an error during evaluation, it prints a message in the following form:

```
Error: text
```

In this message, *text* describes the error. However, if the ***error*** function is defined (that is, if it is not `nil`), AutoLISP executes ***error*** instead of printing the message. The ***error*** function receives *text* as its single argument.

If ***error*** is not defined or is `nil`, AutoLISP evaluation stops and displays a traceback of the calling function and its callers. It is beneficial to leave this error handler in effect while you debug your program.

A code for the last error is saved in the AutoCAD system variable ERRNO, where you can retrieve it by using the **getvar** function. See Error Handling in AutoLISP (page 38) for a list of error codes and their meaning.

Before defining your own ***error*** function, 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. Your ***error*** function should be designed to exit quietly after an ESC (cancel) or an **exit** function call. The standard way to accomplish this is to include the following statements in your error-handling routine.

```
(if
  (or
    (= msg "Function cancelled")
    (= msg "quit / exit abort")
  )
  (princ)
  (princ (strcat "\nError: " msg))
)
```

This code examines the error message passed to it and ensures that the user is informed of the nature of the error. 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 you have already explained the nature of the error by using print statements. Remember to include a terminating call to **princ** if you don't want a return value printed at the end of an error routine.

The main caveat about error-handling routines is they are normal AutoLISP functions that can be canceled by the user. Keep them as short and as fast as possible. This will increase the likelihood that an entire routine will execute if called.

You can also warn the user about error conditions by displaying an alert box, which is a small dialog box containing a message supplied by your program. To display an alert box, call the **alert** function.

The following call to **alert** displays an alert box:

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

Catching Errors and Continuing Program Execution

Your program can intercept and attempt to process errors instead of allowing control to pass to ***error***. The **vl-catch-all-apply** function is designed to invoke any function, return a value from the function, and trap any error that may occur. The function requires two arguments: a symbol identifying a function or **lambda** expression, and a list of arguments to be passed to the called function. The following example uses **vl-catch-all-apply** to divide two numbers:

```
(setq catchit (vl-catch-all-apply '/ '(50 5)))
```

```
10
```

The result from this example is the same as if you had used **apply** to perform the division.

The value of **vl-catch-all-apply** is in catching errors and allowing your program to continue execution.

To catch errors with vl-catch-all-apply

1 The following code defines a function named **catch-me-if-you-can**.

```
(defun catch-me-if-you-can (dividend divisor / errobj)

  (setq errobj (vl-catch-all-apply '/ (list dividend
divisor)))
  (if (vl-catch-all-error-p errobj)
    (progn
      (print (strcat "An error occurred: "
                     (vl-catch-all-error-message
errobj)
             )
      )
      (prompt "Do you want to continue? (Y/N) -> ")
      (setq ans (getstring))
      (if (equal (strcase ans) "Y")
        (print "Okay, I'll keep going")
      )
    )
    (print errobj)
  )
  (princ)
)
```

This function accepts two number arguments and uses **vl-catch-all-apply** to divide the first number by the second number. The **vl-catch-all-error-p** function determines whether the return value from **vl-catch-all-apply** is an error object. If the return value is an error object, **catch-me-if-you-can** invokes **vl-catch-all-error-message** to obtain the message from the error object.

2. Load the function.
3. Invoke the function with the following command:

   (catch-me-if-you-can 50 2)

   The function should return 25.
4. Intentionally cause an error condition by invoking the function with the following command:

   (catch-me-if-you-can 50 0)

   The function should issue the following prompt:

   ```
   "An error occurred: divide by zero" Do you want to
   continue? (Y/N) ->
   ```

   If you enter `y`, **catch-me-if-you-can** indicates that it will continue processing.

   Try modifying this example by changing **vl-catch-all-apply** to **apply**. Load and run the example with a divide by zero again. When **apply** results in an error, execution immediately halts and ***error*** is called, resulting in an error message.

Using AutoLISP to Communicate with AutoCAD

AutoLISP® provides various functions for examining the contents of the currently loaded drawing. This chapter introduces these functions and describes how to use them in conjunction with other functions.

Accessing Commands and Services

The query and command functions described in this section provide direct access to AutoCAD® commands and drawing services. Their behavior depends on the current state of the AutoCAD system and environment variables, and

on the drawing that is currently loaded. See ##xref here - Query and Command Functions (app A Utility functions) in AutoLISP Function Synopsis, (page 119) for a complete list of query and command functions.

Command Submission

The **command** function sends an AutoCAD command 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 or as the values themselves (as integer or real values, or as point lists). An empty string (`""`) is equivalent to pressing the Spacebar or Enter on the keyboard.

There are some restrictions on the commands that you can use with the **command** function. See the *AutoLISP Reference* definition of this function for information on these restrictions.

The following code fragment shows 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)
```

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

1. The first call to **command** passes points to the CIRCLE command as strings (draws a circle centered at 0.0,0.0 and passes through 3.0,3.0).
2. The second call passes an integer to the THICKNESS system variable (changes the current thickness to 1.0).
3. The last call 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. This draws an extruded circle centered at (1.0,1.0,3.0) with a radius of 4.5.

Foreign Language Support

If you develop AutoLISP programs that can be used with a foreign language version of AutoCAD, the standard AutoCAD commands and keywords are automatically translated if you precede each command or keyword with an underscore (_).

```
(command "_line" pt1 pt2 pt3 "_c")
```

If you are using the dot prefix (to avoid using redefined commands), you can place the dot and underscore in either order. Both `"._line"` and `"_.line"` are valid.

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.

The `PAUSE` symbol is defined as a string consisting of a single backslash. When you use a backslash (\) in a string, you must precede it by another backslash (\\).

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.

NOTE You can use a backslash instead of the `PAUSE` symbol. However, it is recommended that you always use the `PAUSE` symbol rather than an explicit backslash. Also, 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.

If you issue a transparent command while a **command** function is suspended, the **command** function remains suspended. Therefore, users can 'ZOOM and 'PAN while at a **command** pause. The pause remains in effect until AutoCAD gets valid input, and no transparent command is in progress. For example, the following code begins the CIRCLE command, sets the center point at (5,5), and then pauses to let the user drag the circle's radius. When the user specifies

the desired point (or types in the desired radius), the function resumes, drawing a line from (5,5) to (7,5), as follows:

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

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 TEXTEVAL system variable is nonzero. Otherwise, AutoCAD does not pause for user input but uses the value of the PAUSE symbol (a single backslash) text.

When the **command** function pauses for user input, the function is considered active, so the user cannot enter another AutoLISP expression to be evaluated.

The following is an example of using the PAUSE symbol (the layer NEW_LAY and the block MY_BLOCK must exist in the drawing prior to testing this code):

```
(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 preceding code fragment sets the current layer to NEW_LAY, pauses for user selection of an insertion point for the block MY_BLOCK (which is inserted with *X* and *Y* scale factors of 1), and pauses again for user selection of a rotation angle. The current layer is then 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.

WARNING 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.

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 object itself. To pass such pairs of object and point data by means of the **command** function without the use of a PAUSE, you must first store them as variables. Points can be passed as strings within the **command** function or can be defined outside the function and passed as variables, as shown in the following example. This 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
```

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

1. Draws a circle centered 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 object added to the database. (See Using AutoLISP to Manipulate AutoCAD Objects (page 74) for more discussion of objects and object-handling functions.)
4. Creates a `pt` variable 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 `el` object and by selecting the point specified by `pt`.

Undoing Commands Issued with the command Function

An UNDO group is explicitly created around each command used with the **command** function. If a user enters U (or UNDO) after running an AutoLISP routine, only the last command will be undone. Additional entries of UNDO will step backward through the commands used in that routine. If you want a group of commands to be considered a group (or the entire routine), use the UNDO Begin and UNDO End options.

System and Environment Variables

With the **getvar** and **setvar** functions, AutoLISP applications can inspect and change the value of AutoCAD system variables. These functions use a string to specify the variable name. The **setvar** function specifies a value of the type that the system variable expects. 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.

The following code fragment ensures that subsequent FILLET commands use a radius of at least 1:

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

See the *Command Reference* for a list of AutoCAD system variables and their descriptions.

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

Configuration Control

AutoCAD uses the *acadxx.cfg* file to store configuration information (the *xx* in the file name refers to the AutoCAD release number). The `AppData` section of this file is provided for users and developers to store configuration information pertaining to their applications. The `getcfg` and `setcfg` functions allow AutoLISP applications to inspect and change the value of parameters in the `AppData` section.

Display Control

AutoLISP includes functions for controlling the AutoCAD display in both text and graphics windows. Some functions prompt for, or depend on, input from the AutoCAD user.

The **prompt**, **princ**, **prin1**, and **print** functions are the primary text output functions. These functions were described in the AutoLISP Basics (page 3) chapter, under the heading, Basic Output Functions. (page 16)

See Display Control Functions (page 136) in AutoLISP Function Synopsis, (page 119) for a complete list of display control functions.

Control of Graphics and Text Windows

You can control the display of the Command Window from an AutoLISP application. A call to **textscr** or **textpage** expands the Command Window.

The **redraw** function is similar to the AutoCAD REDRAW command but provides more control over what is displayed. It not only redraws the entire graphics area but can also specify a single object to be redrawn or undrawn (that is, blanked out). If the object is a complex object such as an old-style polyline or a block, **redraw** can draw (or undraw) either the entire object or its header. The **redraw** function can also highlight and unhighlight specified objects.

.

Control of Low-Level Graphics

AutoLISP provides functions that control the low-level graphics and allow direct access to the AutoCAD graphics screen and input devices.

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 color and highlighting. The **grvecs** function draws multiple vectors.

NOTE 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 current hardware configurations. In particular, applications that call **grtext** are not likely to work the same on all configurations unless the developer is very careful to use them as described (see the *Customization Guide*) and to avoid hardware-specific features. Finally, because they are low-level functions, they do almost no error reporting and can alter the graphics screen display unexpectedly (see the following example for a way to fix this).

The following sequence restores the default graphics window display caused by incorrect calls to **grtext**, **grdraw**, or **grvecs**:

```
(grtext)

Restores standard text

(redraw)
```

Getting User Input

Several functions enable an AutoLISP application to prompt the user for input of data. See User Input Functions (page 141) in AutoLISP Function Synopsis, (page 119) for a complete list of user input functions.

The getxxx Functions

Each user-input **get*xxx*** function pauses for data entry of the indicated type and returns the value entered. The application specifies an optional prompt to display before the function pauses. The following table lists the **get*xxx*** functions and the type of user input requested.

Allowable input to the get*xxx* user-input functions

Function name	Type of user input
getint	An integer value on the command line
getreal	A real or integer value on the command line
getstring	A string on the command line
getpoint	A point value on the command line or selected from the screen
getcorner	A point value (the opposite corner of a box) on the command line or selected from the screen
getdist	A real or integer value (of distance) on the command line or determined by selecting points on the screen

Allowable input to the get*xxx* user-input functions	
Function name	**Type of user input**
getangle	An angle value (in the current angle format) on the command line or based on selected points on the screen
getorient	An angle value (in the current angle format) on the command line or based on selected points on the screen
getkword	A predefined keyword or its abbreviation on the command line

NOTE Although the **getvar**, **getcfg**, and **getenv** functions begin with the letters *g*, *e*, and *t*, they are not user-input functions. They are discussed in Accessing Commands and Services (page 42).

The functions **getint**, **getreal**, and **getstring** pause for user input on the AutoCAD command line. They return a value only of the same type as that requested.

The **getpoint**, **getcorner**, and **getdist** functions pause for user input on the command line or from points selected on the graphics screen. The **getpoint** and **getcorner** functions return 3D point values, and **getdist** returns a real value.

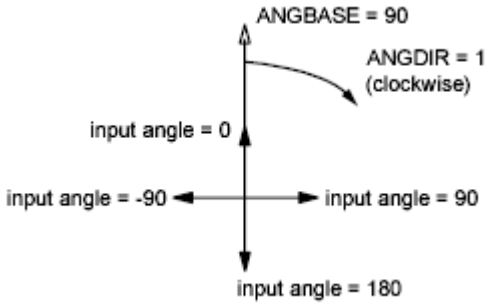
Both **getangle** and **getorient** pause for input of an angle value on the command line or as defined by points selected on the graphics screen. For the **getorient** function, the 0 angle is always to the right: "East" or "3 o'clock." For **getangle**, the 0 angle is the value of ANGBASE, which can be set to any angle. Both **getangle** and **getorient** return an angle value (a real) in radians measured counterclockwise from a base (0 angle), for **getangle** equal to ANGBASE, and for **getorient** to the right.

For example, 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).

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 honors the settings of ANGDIR and ANGBASE when accepting input. You can use **getangle** to obtain a rotation amount for a block insertion, because input of 0 degrees always returns 0 radians. The **getorient** function honors only ANGDIR. You use **getorient** to obtain angles such as the baseline angle for a text object. For example, given the preceding settings of ANGBASE and ANGDIR, for a line of text created at an angle of 0, **getorient** returns an angle value of 90.

The user-input functions take advantage of the error-checking capability of AutoCAD. Trivial errors are trapped by AutoCAD and are not returned by the user-input function. A prior call to **initget** provides additional filtering capabilities, lessening the need for error-checking.

The **getkword** function pauses for the 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**) can accept keyword values in addition to the values they normally return, provided that **initget** has been called to define the keywords.

All user-input functions allow for an optional *prompt* argument. It is recommended you use this argument rather than a prior call to the **prompt** or **princ** functions. 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, because the original prompt may have scrolled out of the Command prompt area.

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 a command implemented in AutoLISP. This is discussed in Arbitrary Keyboard Input (page 54).

Control of User-Input Function Conditions

The **initget** function provides a level of control over the next user-input function call. The **initget** function establishes various options for use by the next **entsel**, **nentsel**, **nentselp**, or **get*xxx*** 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 next user-input function call. The *string* argument can specify keywords that the next user-input function call will recognize.

The control bits and keywords established by **initget** apply only to the next user-input function call. They are discarded after that call. The application doesn't have to call **initget** a second time to clear special conditions.

Input Options for User-Input Functions

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

To set more than one condition at a time, 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 0, none of the control conditions applies to the next user-input function call. (For a complete listing of **initget** bit settings, see **initget** in the *AutoLISP Reference*.)

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

This sequence requests the user's age. AutoCAD displays an error message and repeats the prompt if the user attempts to enter a negative or zero value, or if the user only presses Enter, or enters a string (the **getint** function rejects attempts to enter a value that is not an integer).

Keyword Options

The optional *string* argument specifies a list of keywords recognized by the next user-input function call.

The **initget** function allows keyword abbreviations to be recognized in addition to the full keywords. 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. There are two methods for abbreviating keywords; both are discussed in the **initget** topic in the *AutoLISP Reference*.

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 following 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 p 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. For more information on these functions, see Object Handling (page 86) and the **entsel**, **nentsel** and **nentselp** function definitions in the *AutoLISP Reference*.

Arbitrary Keyboard Input

The **initget** function also allows arbitrary keyboard input to most **get*xxx*** 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 **get*xxx*** function prompt.

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

```
(defun C:ARBENTRY ( / pt1)
  (initget 128)                     ; Sets arbitrary entry
 bit
  (setq pt1 (getpoint "\nPoint: ")) ; Gets value from user.

  (if (= 'STR (type pt1))           ; If it's a string,
convert it
    (setq pt1 (eval (read pt1)))    ; to a symbol, try
evaluating
                                    ; it as a function;
otherwise,
```

```
      pt1                              ; just return the value.

    )
  )

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

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

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

Input Validation

You should protect your code from unintentional user errors. The AutoLISP user input **get*xxx*** functions do much of this for you. However, it's dangerous to forget to check for adherence to other program requirements that the **get*xxx*** functions do not check for. If you neglect to check input validity, the program's integrity can be seriously affected.

Geometric Utilities

A group of functions allows applications to obtain pure geometric information and geometric data from the drawing. See Geometric Functions (page 139) in AutoLISP Function Synopsis, (page 119) for a complete list of geometric utility functions.

The **angle** function finds the angle in radians between a line and the *X* axis (of the current UCS), **distance** finds the distance between two points, and **polar** finds a point by means of polar coordinates (relative to an initial point). The **inters** function finds the intersection of two lines. The **osnap** and **textbox** functions are described separately.

The following code fragment shows 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`.

Object Snap

The osnap function can find a point by using one of the AutoCAD Object Snap modes. The Snap modes are specified in a string argument.

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

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

The following call looks for the midpoint, the endpoint, or the center of an object 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. If more than one snap point fulfills the requirements, the point is selected based on the setting of the SORTENTS system variable. Otherwise, `pt2` is set to `nil`.

NOTE The APERTURE system variable determines the allowable proximity of a selected point to an object when you use Object Snap.

Text Extents

The **textbox** function returns the diagonal coordinates of a box that encloses a text object. It takes an entity definition 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 object or just a list describing the text string.

The points returned by **textbox** describe the bounding box (an imaginary box that encloses the text object) of the text object, as if its insertion point were located at (0,0,0) and its rotation angle were 0. The first list returned is the point (0.0 0.0 0.0), unless the text object is oblique or vertical or it contains letters with descenders (such as g and p). The value of the first point list specifies the offset distance 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.

The following example shows the minimum allowable entity definition list that **textbox** accepts. Because 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))
```

The actual values returned by **textbox** will vary depending on the current text style.

The following example demonstrates one method of providing the **textbox** function with an entity definition list.

```
Command: dtext
Justify/Style/<Start point>: 1,1
Height <1.0000>: Enter
Rotation angle <0>: Enter
Text: test
Text: Enter
Command: (setq e (entget (entlast)))
((-1 . <Entity name: 1ba3568>) (0 . "TEXT") (330 . <Entity
name: 1ba34f8>) (5 .
"2D") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0")
(100 .
"AcDbText") (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)
(100 . "AcDbText") (73 . 0))
Command: (textbox e)
((0.0 0.0 0.0) (0.8 0.2 0.0))
```

The following figure shows the results of applying **textbox** to a text object with a height of 1.0. The figure also shows the baseline and insertion point of the text.

If the text is vertical or rotated, `pt1` is still the bottom-left corner and `pt2` is the upper-right corner; the bottom-left point may have negative offsets if necessary.

The following figure 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.)

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 that 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 object coordinate system (OCS) for the associated text object. This point can be referenced when translating the coordinates returned from **textbox** into points that define the actual extent of the text. The two sample routines that follow use **textbox** to place a box around selected text regardless of its orientation.

The first routine uses the **textbox** function to draw a box around a selected text object:

```
(defun C:TBOX ( / textent tb ll ur ul lr)
  (setq textent (car (entsel "\nSelect text: ")))
  (command "ucs" "Object" 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)
)
```

The second routine, which follows, accomplishes the same task as the first 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 object.

```
(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)
)
```

Conversions

The functions described in this section are utilities for converting data types and units. See in AutoLISP Function Synopsis, (page 119) for a complete list of conversion functions.

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 calls to **rtos** and the values returned (assuming the DIMZIN system variable equals 0). 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))    ;

displays
Value formatted as 1.7500E+01

(setq fmtval (rtos x 2 2))     ; Mode 2 = decimal
(princ (strcat str fmtval))    ;

displays
Value formatted as 17.50
```

```
(setq fmtval (rtos x 3 2))     ; Mode 3 = engineering
(princ (strcat str fmtval))    ;

displays
Value formatted as 1'-5.50"

(setq fmtval (rtos x 4 2))     ; Mode 4 = architectural
(princ (strcat str fmtval))    ;

displays
Value formatted as 1'-5 1/2"

(setq fmtval (rtos x 5 2))     ; Mode 5 = fractional
(princ (strcat str fmtval))    ;

displays
Value formatted as 17 1/2
```

When the UNITMODE system variable is set to 1, specifying that units are displayed as 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 preceding 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''
```

Because the **angtos** function takes the ANGBASE system variable into account, the following code always returns `"0"`:

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

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:

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

Subtracting the result of **(atof (angtos 0))** from 360 degrees (2pi; radians or 400 grads) also yields the rotation of ANGBASE from 0.

The **distof** (distance to floating point) function is the complement of **rtos**. Therefore, 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 0). 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))     ;

displays
Angle formatted as 180

(setq fmtval (angtos ang 1 4))   ; Mode 1 = deg/min/sec
(princ (strcat str2 fmtval))     ;

displays
Angle formatted as 180d0'0"

(setq fmtval (angtos ang 2 4))   ; Mode 2 = grads
(princ (strcat str2 fmtval))     ;

displays
Angle formatted as 200.0000g

(setq fmtval (angtos ang 3 4))   ; Mode 3 = radians
(princ (strcat str2 fmtval))     ;

displays
Angle formatted as 3.1416r

(setq fmtval (angtos ang 4 2))   ; Mode 4 = surveyor's
```

```
(princ (strcat str2 fmtval))     ;

displays
Angle formatted as W
```

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 0, the string returned can include spaces (for example, `"N 45d E"`); if `UNITMODE` equals 1, the string contains no spaces (for example, `"N45dE"`).

The **angtof** function complements **angtos**, so all of the following calls return 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
```

When you have a string specifying a distance in feet and inches, or an angle in degrees, minutes, and seconds, you must precede the quotation mark with a backslash (\") so it doesn't look like the end of the string. The preceding examples of **angtof** and **distof** demonstrate this action.

Angular Conversion

If your application needs to convert angular values from radians to degrees, you can use the **angtos** function, which returns a string, and then convert that string into a floating point value with **atof**.

```
(setq pt1 '(1 1) pt2 '(1 2))
(setq rad (angle pt1 pt2))
(setq deg (atof (angtos rad 0 2)))

returns
90.0
```

However, a more efficient method might be to include a **Radian->Degrees** function in your application. The following code shows this:

```
; Convert value in radians to degrees
(defun Radian->Degrees (nbrOfRadians)
  (* 180.0 (/ nbrOfRadians pi))
)
```

After this function is defined, you can use the **Radian->Degrees** function throughout your application, as in

```
(setq degrees (Radian->Degrees rad))

returns
90.0
```

You may also need to convert from degrees to radians. The following code shows this:

```
; Convert value in degrees to radians
(defun Degrees->Radians (numberOfDegrees)
  (* pi (/ numberOfDegrees 180.0))
) ;_ end of defun
```

ASCII Code Conversion

AutoLISP provides the **ascii** and **chr** functions that handle decimal ASCII codes. The **ascii** function returns the ASCII decimal value associated with a string, and **chr** returns the character associated with an ASCII decimal value.

To see your system's characters with their codes in decimal, octal, and hexadecimal form, save the following AutoLISP code to a file named *ascii.lsp*. Then load the file and enter the new ASCII command at the AutoCAD Command prompt. This command prints the ASCII codes to the screen and to a file called *ascii.txt*. The **C:ASCII** function makes use of the **BASE** function. You may find this conversion utility useful in other applications.

```
; BASE converts from a decimal integer to a string in
another base.
(defun BASE ( bas int / ret yyy zot )
  (defun zot ( i1 i2 / xxx )
    (if (> (setq xxx (rem i2 i1)) 9)
      (chr (+ 55 xxx))
      (itoa xxx)
    )
  )
  (setq ret (zot bas int) yyy (/ int bas))
  (while (>= yyy bas)
    (setq ret (strcat (zot bas yyy) ret))
    (setq yyy (/ yyy bas))
  )
  (strcat (zot bas yyy) ret)
```

```
)

(defun C:ASCII ( / chk out ct code dec oct hex )
  (initget "Yes")
  (setq chk (getkword "\nWriting to ASCII.TXT, continue?
<Y>: "))
  (if (or (= chk "Yes")(= chk nil))    (progn
      (setq out (open "ascii.txt" "w") chk 1 code 0 ct 0)
      (princ "\n \n CHAR   DEC  OCT  HEX \n")
      (princ "\n \n CHAR   DEC  OCT  HEX \n" out)
      (while chk
        (setq dec (strcat "  " (itoa code))
          oct (base 8 code) hex (base 16 code))
        (setq dec (substr dec (- (strlen dec) 2) 3))
        (if (< (strlen oct) 3)(setq oct (strcat "0" oct)))

        (princ (strcat "\n " (chr code) "      " dec " "
          oct "  " hex ) )
        (princ (strcat "\n " (chr code) "      " dec " "
          oct "  " hex ) out)
        (cond
          ((= code 255)(setq chk nil))
          ((= ct 20)
            (setq xxx (getstring
               "\n \nPress 'X' to eXit or any key to
continue: "))
            (if (= (strcase xxx) "X")
              (setq chk nil)
              (progn
                (setq ct 0)
                (princ "\n \n CHAR   DEC  OCT  HEX \n")
              )
            )
          )
        )
        (setq ct (1+ ct) code (1+ code))
      )
      (close out)
      (setq out nil)
    )
  )
  (princ)
)
```

Unit Conversion

The *acad.unt* file defines various conversions between real-world units such as miles to kilometers, Fahrenheit to 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*.

The **cvunit** function does not convert incompatible dimensions. For example, it does not convert inches into grams.

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** and 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.

Converting from Inches to Meters

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

```
(defun C:I2M ( / eng_len metric_len eng metric)
  (princ "\nConverting inches to meters. ")
  (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 " meters."))
  (princ)
)
```

The Unit Definition File

With the AutoCAD unit definition file *acad.unt*, you can define factors to convert data in one set of units to another set of units. The definitions in

acad.unt are in ASCII format and are used by the unit-conversion function **cvunit**.

You can make new units available by using a text editor to add their definitions to *acad.unt*. A definition consists of two lines in the file—the unit name and the unit definition. The first line must have an asterisk (*) in the first column, followed by the name of the unit. A unit name can have several abbreviations or alternate spellings, separated by commas. If a unit name has singular and plural forms, you can specify these using the following format:

```
*[ [common] [ ( [singular.] plural) ] ]...
```

You can specify multiple expressions (singular and plural). They don't have to be located at the end of the word, and a plural form isn't required. The following are examples of valid unit name definitions:

```
*inch(es)
*milleni(um.a)
*f(oot.eet) or (foot.feet)
```

The line following the ***unit name* line defines the unit as either fundamental or derived.

Fundamental Units

A fundamental unit is an expression in constants. If the line following the ***unit name* line begins with something other than an equal sign (=), it defines fundamental units. Fundamental units consist of five integers and two real numbers in the following form:

```
c, e, h, k, m, r1, r2
```

The five integers correspond to the exponents of these five constants:

c Velocity of light in a vacuum

e Electron charge

h Planck's constant

k Boltzman's constant

m Electron rest mass

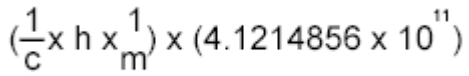
As a group, these exponents define the dimensionality of the unit: length, mass, time, volume, and so on.

The first real number (r1) is a multiplier, and the second (r2) is an additive offset (used only for temperature conversions). The fundamental unit definition allows for different spellings of the unit (for example, *meter* and *metre*); the

case of the unit is ignored. An example of a fundamental unit definition is as follows:

```
*meter(s),metre(s),m
-1,0,1,0,-1,4.1214856408e11,0
```

In this example, the constants that make one meter are as follows:



Derived Units

A derived unit is defined in terms of other units. If the line following the ***unit name*** line begins with an equal sign (=), it defines derived units. Valid operators in these definitions are `*` (multiplication), `/` (division), `+` (addition), `-` (subtraction), and `^` (exponentiation). You can specify a predefined unit by naming it, and you can use abbreviations (if provided). The items in a formula are multiplied together unless some other arithmetic operator is specified. For example, the units database defines the dimensionless multiple and submultiple names, so you can specify a unit such as micro-inches by entering **micro inch**. The following are examples of derived unit definitions.

```
; Units of area
*township(s)
=93239571.456 meter^2
```

The definition of a township is given as 93,239,571.456 square meters.

```
; Electromagnetic units
*volt(s),v
=watt/ampere
```

In this example, a volt is defined as a watt divided by an ampere. In the *acad.unt*, both watts and amperes are defined in terms of fundamental units.

User Comments

To include comments, begin the line with a semicolon. The comment continues to the end of the line.

```
; This entire line is a comment.
```

List the *acad.unt* file itself for more information and examples.

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 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 is the syntax for the **trans** function:

```
(trans pt from to [disp])
```

The following AutoCAD coordinate systems 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 drawing tasks easier. 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, OCS, or DCS to AutoCAD commands, you must first convert them to the UCS by calling the **trans** function.

OCS Object coordinate system—point values returned by **entget** are expressed in this coordinate system, relative to the object itself. These points are usually converted into the WCS, current UCS, or current DCS, according to the intended use of the object. Conversely, points must be translated into an OCS before they are written to the database by means of the **entmod** or **entmake** functions. This is also known as the entity coordinate system.

DCS Display coordinate system—the coordinate system into which objects are transformed before they are 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 will be displayed to the AutoCAD user.

When the *from* and *to* integer codes are 2 and 3, in either order, 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), it is assumed to indicate the DCS of the current space, whether paper space or model space. 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 0. The *Z* coordinate is scaled but is never translated. Therefore, it can be used to find the scale factor between the two coordinate systems. The PSDCS (integer code 2) can be transformed only 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:

- As an integer code that specifies the WCS, current UCS, or current DCS (of either the current viewport or paper space).
- As an entity name returned by one of the entity name or selection set functions described in Using AutoLISP to Manipulate AutoCAD Objects. (page 74) This specifies the OCS of the named object. For planar objects, the OCS can differ from the WCS, as described in the AutoCAD*User's Guide*. If the OCS does not differ, conversion between OCS and WCS is an identity operation.
- As a 3D extrusion vector. Extrusion vectors are always represented in World coordinates; an extrusion vector of (0,0,1) specifies the WCS itself.

The following table lists the valid integer codes that can be used as the *to* and *from* arguments:

Coordinate system codes

Code	Coordinate system
0	World (WCS)
1	User (current UCS)

Coordinate system codes	
Code	**Coordinate system**
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)

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
(trans pt cs_from cs_to 0)    ;

disp = 0 indicates that pt is a point
```

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). However, if you swap the *to* and *from* values, the result differs as shown in the following code:

```
(trans pt cs_to cs_from 0) ;

the result is (-2.0,1.0,3.0)
```

Point Transformations

If you are doing point transformations with the **trans** function and you 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). Writing matrix multiplication functions in AutoLISP can be difficult, so it may not be worthwhile unless your program is doing a lot of transformations.

File Handling

AutoLISP provides functions for handling files and data I/O. See File-Handling Functions (page 137) in AutoLISP Function Synopsis, (page 119) for a complete list of file-handling functions.

File Search

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

In the following code fragment, **findfile** searches for the requested file name 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 path name string, as follows:

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

The **getfiled** function displays a dialog 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 dialog box.

A call to **getfiled** takes four arguments that determine the appearance and functionality of the dialog box. The application must specify the following string values, each of which can be `nil`: a title, placed at the top of the dialog box; a default file name, displayed in the edit box at the bottom of the dialog box; 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 dialog box interacts with selected files.

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 is a useful utility command. The `dfil` variable 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 file name. To use this variable in response to a command line prompt, enter **!dfil**.

NOTE You cannot use **!dfil** in a dialog box. It is valid only at the command line.

For more information, see **getfiled** in the *AutoLISP Reference*.

Device Access and Control

AutoLISP provides the **grread** and **tablet** functions for accessing data from the various input devices.

Note that the **read-char** and **read-line** file-handling functions can also read input from the keyboard input buffer. See the *AutoLISP Reference* for more information on these functions.

Accessing User Input

The **grread** function returns raw user input, whether from the keyboard or from the pointing device (mouse or digitizer). If the call to **grread** enables tracking, the function returns a digitized coordinate that can be used for things such as dragging.

NOTE There is no guarantee that applications calling **grread** will be upward compatible. Because it depends on the current hardware configuration, applications that call **grread** are not likely to work in the same way on all configurations.

Using AutoLISP to Manipulate AutoCAD Objects

Most AutoLISP® functions that handle selection sets and objects identify a set or an object by the entity name. For selection sets, which are valid only in the current session, the volatility of names poses no problem, but it does for objects because they are saved in the drawing database. An application that must refer to the same objects in the same drawing (or drawings) at different times can use the objects' handles.

AutoLISP uses symbol tables to maintain lists of graphic and non-graphic data related to a drawing, such as the layers, linetypes, and block definitions. Each symbol table entry has a related entity name and handle and can be

manipulated in a manner similar to the way other AutoCAD® entities are manipulated.

Selection Set Handling

AutoLISP provides a number of functions for handling selection sets. For a complete list of selection set functions, see Selection Set Manipulation Functions (page 145) in AutoLISP Function Synopsis, (page 119)

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 objects to select, using the Last, Previous, Window, Implied, WPolygon, Crossing, CPolygon, or Fence options
- Specifying a single point
- Selecting the entire database
- Prompting the user to select objects

With any option, you can use filtering to specify a list of attributes and conditions that the selected objects must match.

NOTE Selection set and entity names are volatile. That is, they apply only to the current drawing session.

The first argument to **ssget** is a string that describes 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 (that is, 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 used in filtering. For example, you can obtain a selection set that includes all objects of a given type, on a given layer, or of a given color. Selection filters are described in more detail in Selection Set Filter Lists (page 77).

See the ssget entry in the *AutoLISP Reference* for a list of the available selection methods and the arguments used with each.

The following table shows examples of calls to **ssget**:

SSGET Examples	
Function call	**Effect**

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

When an application has finished using a selection set, it is important to release it from memory. You can do this by setting it to `nil`:

```
(setq ss1 nil)
```

Attempting to manage a large number of selection sets simultaneously is not recommended. An AutoLISP application cannot have more than 128 selection sets open at once. (The limit may be lower on your system.) When the limit is reached, AutoCAD will not create more selection sets. Keep a 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 the gc function to free unused memory before another **ssget** will work.

Selection Set Filter Lists

An entity filter list is an association list that uses DXF group codes in the same format as a list returned by **entget**. (See the *DXF Reference* for a list of group codes.) The **ssget** function recognizes all group codes except entity names (group -1), handles (group 5), and xdata codes (groups greater than 1000). If an invalid group code is used in a *filter-list*, it is ignored by **ssget**. To search for objects with xdata, use the -3 code as described in Filtering for Extended Data (page 80).

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

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

The following examples demonstrate methods of using a *filter-list* with various object selection options.

SSGET examples using filter lists

Function call	Effect
`(setq ss1 (ssget '((0 . "TEXT")))` `)`	Prompts for general object selection but adds only text objects to the selection set.

SSGET examples using filter lists

Function call	Effect

`(setq ss1 (ssget "P"`
`'((0 . "LINE")))`
`)`
	Creates a selection set containing all line objects from the last selection set created.

`(setq ss1 (ssget "W" pt1 pt2`
`'((8 . "FLOOR9")))`
`)`
	Creates a selection set of all objects inside the window that are also on layer FLOOR9.

`(setq ss1 (ssget "X"`
`'((0 . "CIRCLE")))`
`)`
	Creates a selection set of all objects in the database that are Circle objects.

`(ssget "I" '((0 . "LINE")` `(62 . 5)))`	Creates a selection set of all blue Line objects that are part of the Implied selection set (those objects selected while PICKFIRST is in effect). Note that this filter picks up lines that have been assigned color 5 (blue), but not blue lines that have had their color applied by the ByLayer or ByBlock properties.

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** function. For example, the following code creates a selection set of all objects in the database that are on layer FLOOR3:

```
(setq lay_name "FLOOR3")
(setq ss1
  (ssget "X"
    (list (cons 8 lay_name))
  )
)
```

If the *filter-list* specifies more than one property, an entity is included in the selection set only if it matches all specified conditions, as in the following example:

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

This code selects only Circle objects on layer FLOOR3 that are colored red. This type of test performs a Boolean "AND" operation. Additional tests for object properties are described in Logical Grouping of Filter Tests (page 82).

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.

NOTE 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 are excluded from the selection set that **ssget** returns.

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 is currently in effect are used. (The space to which an entity belongs is specified by the value of its 67 group. Refer to the *Customization Guide* for further information.)

Wild-Card Patterns in Filter Lists

Symbol names specified in filtering lists can include wild-card patterns. The wild-card patterns recognized by **ssget** are the same as those recognized by the **wcmatch** function, and are described in Wild-Card Matching (page 20), and under wcmatch in the *AutoLISP Reference*.

When filtering for anonymous blocks, you must precede the `*` character with a reverse single quotation mark (`), also known as an escape character, because the `*` is read by **ssget** as a wild-card character. For example, you can retrieve an anonymous block named `*U2` with the following:

```
(ssget "X" '((2 . "`*U2")))
```

Filtering for Extended Data

Using the **ssget***filter-list*, you can select all entities containing extended data for a particular application. (See Extended Data—xdata (page 106).) To do this, use the -3 group code, as shown in the following example:

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

This code will 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. So, the following statement would select all circles with extended data for both the "APP1" and "APP2" applications:

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

Wild-card matching is permitted, so either of the following statements will select all circles with extended data for either or both of these applications.

```
(ssget "X" '((0 . "CIRCLE") (-3 ("APP[12]"))))
(ssget "X" '((0 . "CIRCLE") (-3 ("APP1,APP2"))))
```

Relational Tests

Unless otherwise specified, an equivalency 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*.

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

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

The possible relational operators are shown in the following table:

Relational operators for selection set filter lists

Operator	Description
"*"	Anything goes (always true)
"="	Equals

Relational operators for selection set filter lists

Operator	Description
"!="	Not equal to
"/="	Not equal to
"<>"	Not equal to
"<"	Less than
"<="	Less than or equal to
">"	Greater than
">="	Greater than or equal to
"&"	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 (for example, ">,>,*"). If an operator is omitted from the string (for example, "=,<>" leaves out the *Z* test), then 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 equal" strings).

- You cannot use the relational operators with string groups; use wild-card tests instead.

Logical Grouping of Filter Tests

You can also test groups by creating nested Boolean expressions that use the logical grouping operators shown in the following table:

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>"

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 code selects all circles with a radius of 1.0 plus all lines on layer "ABC". The grouping operators are not case-sensitive; for example, you can specify "and>", "<or", instead of "**AND>**", "<OR".

Grouping operators are not allowed within the -3 group. Multiple application names specified in a -3 group use an implied AND operator. If you want to test for extended data using other grouping operators, specify separate -3 groups and group them as desired. To select all circles having extended data for either application "APP1" or "APP2" but not both, enter the following:

```
(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 may 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, you can add entities to it or remove entities from it with the functions **ssadd** and **ssdel**. You can use the **ssadd** function to create a new selection set, as shown in the following 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 selection set
 of the
                                 ; first entity.
    (ssadd lname ourset)         ; Adds the last entity to
 the
                                 ; selection set.
  )
)
```

The example runs correctly even if only one entity is 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 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), then the selection set `ourset` 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 to the set (entities in a selection set are numbered from 0).

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

```
(setq sset (ssget))              ; Prompts the user to create
 a
                                 ; selection set.
(setq ent1 (ssname sset 0))      ; Gets the name of the 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)))
```

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

Passing Selection Sets between AutoLISP and ObjectARX Applications

When passing selection sets between AutoLISP and ObjectARX applications, the following should be observed:

If a selection set is created in AutoLISP and stored in an AutoLISP variable, then overwritten by a value returned from an ObjectARX application, the original selection set is eligible for garbage collection (it is freed at the next automatic or explicit garbage collection).

This is true even if the value returned from the ObjectARX application was the original selection set. In the following example, if the **adsfunc** ObjectARX function returns the same selection set it was fed as an argument, then this selection set will be eligible for garbage collection even though it is still assigned to the same variable.

```
(setq var1 (ssget))
(setq var1 (adsfunc var1))
```

If you want the original selection set to be protected from garbage collection, then you must not assign the return value of the ObjectARX application to the AutoLISP variable that already references the selection set. Changing the previous example prevents the selection set referenced by `var1` from being eligible for garbage collection.

```
(setq var1 (ssget))
(setq var2 (adsfunc var1))
```

Object Handling

AutoLISP provides functions for handling objects. The object-handling functions are organized into two categories: functions that retrieve the entity name of a particular object, and functions that retrieve or modify entity data. See Object-Handling Functions (page 143) in AutoLISP Function Synopsis, (page 119) for a complete list of the object-handling functions.

Entity Name Functions

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

Both functions require the AutoCAD user to select an object interactively by picking a point on the graphics screen. All the other entity name functions can retrieve an entity even if it is not visible on the screen or if it is on a frozen layer. The **entsel** function prompts the user to select an object by picking a point on the graphics screen, and **entsel** returns both the entity name and the value of the point selected. Some entity operations require knowledge of the point by which the object was selected. Examples from the set of existing AutoCAD commands include: BREAK, TRIM, and EXTEND. The **nentsel** function is described in detail in Entity Context and Coordinate Transform Data (page 88). These functions accept keywords if they are preceded by a call to **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, **entnext** returns the name of the succeeding entity.

The following 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 with a call to **command**.

You can set the entity name returned by **entnext** to the same variable name passed to this function. This "walks" a single entity name variable through the database, as shown in the following 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 Uses

The **handent** function retrieves the name of an entity with a specific handle. As with entity names, handles are unique within a drawing. However, an entity's handle is constant throughout its life. AutoLISP applications that manipulate a specific database can use **handent** to obtain the current name of an entity they must use. You can use the LIST command to get the handle of a selected object.

The following code fragment uses **handent** to obtain and display an entity name.

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

In one particular editing session, this code fragment might display the following:

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

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

The **handent** function has an additional use. Entities can be deleted from the database with **entdel** (see Entity Context and Coordinate Transform Data (page 88)). The entities are not 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 Handles are provided for block definitions, including subentities.

Entities in drawings that are cross-referenced by way of 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 by means of INSERT, INSERT *, XREF Bind (XBIND), or partial DXFIN, the handles of entities in the incoming drawing are lost, and incoming entities are assigned new handle values to ensure each handle in the current drawing remains unique.

Entity Context and Coordinate Transform Data

The **nentsel** and **nentselp** functions are similar to **entsel**, except they return two additional values to handle entities nested within block references.

Another difference between these functions is that when the user responds to a **nentsel** call by selecting 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 selects a 3D polyline, **nentsel** returns a vertex subentity instead of the polyline header. To retrieve the polyline header, the application must use **entnext** to step forward to the seqend subentity, and 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.

Selecting an attribute within a block reference returns the name of the attribute and the pick point. When the selected object is a component of a block reference other than an attribute, **nentsel** returns a list containing the following elements:

- The selected entity's name.
- A list containing the coordinates of the point used to pick the object.
- The Model to World Transformation Matrix. This 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.
- A list containing the entity name of the block that contains the selected object. If the selected object is in a nested block (a block within a block), the list also contains the entity names of all blocks in which the selected object is nested, starting with the innermost block and continuing outward until the name of the block that was inserted in the drawing is reported.

The list returned from selecting a block with **nentsel** is summarized as follows:

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

In the following example, create a block to use with the **nentsel** function.

```
Command: line
Specify first point: 1,1
Specify next point or [Undo]: 3,1
Specify next point or [Undo]: 3,3
Specify next point or [Close/Undo]: 1,3
Specify next point or [Close/Undo]: c
Command: -block
Enter block name or [?]: square
Specify insertion base point: 2,2
Select objects: Select the four lines you just drew
Select objects: Enter
```

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

```
Command: ucs
Current ucs name: *WORLD*
Enter
option[New/Move/orthoGraphic/Prev/Restore/Save/Del/Apply/?/World]
<World>: z
Specify rotation angle about Z axis <0>: 45
Command: -insert
Enter block name or [?]: square
Specify insertion point or
[Scale/X/Y/Z/Rotate/PScale/PX/PY/PZ/PRotate]:7,0
Enter X scale factor, specify opposite corner, or [Corner/XYZ]
<1>: Enter
Enter Y scale factor <use X scale factor>: Enter
Specify rotation angle <0>: Enter
```

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

(setq ndata (nentsel))

This code sets `ndata` equal to a list similar to the following:

```
(<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 object.
)
```

Once you obtain the entity name and the Model to World Transformation Matrix, 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 definition points expressed in MCS coordinates. The Model to World Transformation Matrix returned by **nentsel** 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:

$$\begin{bmatrix} X' & Y' & Z' & 1.0 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1.0 \end{bmatrix} \bullet \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \\ M_{30} & M_{31} & M_{32} \end{bmatrix}$$

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 *Mij*, where 0 le; *i*, *j le;* 2, are the Model to World Transformation Matrix coordinates; *X*, *Y*, *Z* is the entity definition data point expressed in MCS coordinates, and *X*', *Y*', *Z*' is the resulting entity definition data point expressed in WCS coordinates.

To transform a vector rather than a point, do not add the translation vector (M30 M31 M32 from the fourth column of the transformation matrix).

NOTE This is the only AutoLISP function that uses a matrix of this type. The **nentselp** function is preferred to **nentsel** because it returns a matrix similar to those used by other AutoLISP and ObjectARX functions .

Using the entity name previously obtained with **nentsel**, the following example illustrates how to obtain the MCS start point of a line (group code 10) contained in a block definition:

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

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

The following statement stores the Model to World Transformation Matrix sublist in the symbol`matrix`.

`Command:` **(setq matrix (caddr ndata))**

```
((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
)
```

The following command applies 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:

```
(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
  )
)
```

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

Entity Access Functions

The entity access functions are relatively slow. It is best to get the contents of a particular entity (or symbol table entry) once and keep that information stored in memory, rather than repeatedly ask AutoCAD for the same data. Be sure the data remains valid. If the user has an opportunity to alter the entity or symbol table entry, you should reissue the entity access function to ensure the validity of the data.

Entity Data Functions

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

Deleting an Entity

The **entdel** function deletes a specified entity. The entity is not 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.

Attributes and old-style polyline vertices cannot be deleted independently of their parent entities. The **entdel** function 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.

Obtaining Entity Information

The **entget** function returns the definition data of a specified entity. The data is 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.

In this example, the following (default) conditions apply to the current drawing:

- Layer is 0
- Linetype is CONTINUOUS
- Elevation is 0

The user has drawn a line with the following sequence of commands:

```
Command: line
From point: 1,2
To point: 6,6
To point: Enter
```

An AutoLISP application can retrieve and print the definition data for the line by using the following AutoLISP function:

```
(defun C:PRINTDXF ( )
  (setq ent (entlast))     ; Set ent to last entity.
  (setq entl (entget ent)) ; Set entl to association list
 of
                           ; last entity.
  (setq ct 0)              ; Set ct (a counter) to 0.
  (textpage)               ; Switch to the text screen.
  (princ "\nentget of last entity:")
```

```
    (repeat (length entl)    ; Repeat for number of members
   in list:
      (print (nth ct entl))  ; Print a newline, then each
  list
                             ; member.
      (setq ct (1+ ct))     ; Increments the counter by one.

    )
    (princ)                  ; Exit quietly.
  )
```

This would print the following:

```
entget of last entity:
(-1 . <Entity name: 1bbd1c8>)
(0 . "LINE")
(330 . <Entity name: 1bbd0c8>)
(5 . "69")
(100 . "AcDbEntity")
(67 . 0)
(410 . "Model")
(8 . "0")
(100 . "AcDbLine")
(10 1.0 2.0 0.0)
(11 6.0 6.0 0.0)
(210 0.0 0.0 1.0)
```

The -1 item at the start of the list contains the name of the entity. The **entmod** function, which is described in this section, uses the name to identify the entity to be modified. The individual dotted pairs that represent the values can be extracted by using **assoc** with the **cdr** function.

Sublists for points are not represented as dotted pairs like the rest of the values returned. The convention is that the **cdr** of the sublist is the group's value. Because a point is a list of two or three reals, the entire group is 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.

The codes for the components of the entity are those used by DXF. 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 or not they equal their defaults and whether or not 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.

All points associated with an object are expressed in terms of that object's object coordinate system (OCS). For point, line, 3D line, 3D face, 3D polyline, 3D mesh, and dimension objects, the OCS is equivalent to the WCS (the object points are World points). For all other objects, the OCS can be derived from the WCS and the object's extrusion direction (its 210 group). When working with objects that are drawn using coordinate systems other than the WCS, you may need to convert the points to the WCS or to the current UCS by using the **trans** function.

When writing functions to process entity lists, make sure the function logic is independent of the order of the sublists; use **assoc** to guarantee this. The **assoc** function searches a list for a group of a specified type. The following code returns the object type "LINE" (0) from the list `ent1`.

```
(cdr (assoc 0 ent1))
```

If the DXF group code specified is not present in the list (or if it is not a valid DXF group), **assoc** returns `nil`.

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

Modifying an Entity

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, and then passing the list back to the database with **entmod**.

The following 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 first entity name
                        ; 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
                        ; the drawing.
```

There are restrictions on the changes to the database that **entmod** can make; **entmod***cannot* change the following:

- The entity's type or handle.
- 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 ignored.
- Viewport entities. An attempt to change a viewport entity causes an error.

Other restrictions apply when modifying dimensions and hatch patterns.

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**. There is one 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.

The **entmod** function can modify subentities such as polyline vertices and block attributes.

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**.

Adding an Entity to a Drawing

An application can add an entity to the drawing database by calling the **entmake** function. Like that of **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 (an old-style polyline or a block), it is not appended to the database until it is complete.

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

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

The following **entmake** restrictions apply to all entities:

- 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 specify only the name of the entity: group -1 (the name is not saved in the database).
- AutoCAD must recognize all objects that the entity list refers to. There is one exception: **entmake** accepts new layer names.
- Any internal fields passed to **entmake** are ignored.
- **entmake** cannot create viewport entities.

For entity types introduced in AutoCAD Release 13 and later releases, you must also specify subclass markers (DXF group code 100) when creating the entity. All AutoCAD entities have the AcDbEntity subclass marker, and this must be explicitly included in the **entmake** list. In addition, one or more subclass marker entries are required to identify the specific sub-entity type. These entries must follow group code 0 and must precede group codes that are specifically used to define entity properties in the **entmake** list. For example, the following is the minimum code required to **entmake** an MTEXT entity:

```
(entmake '(
 (0 . "MTEXT")
 (100 . "AcDbEntity")   ; Required for all post-R12
entities.
 (8 . "ALAYER")
 (100 . "AcDbMText")    ; Identifies the entity as MTEXT.
 (10 4.0 4.0 0.0)
 (1 . "Some\\Ptext")
 )
)
```

The following table identifies the entities that do not require subentity marker entries in the list passed to **entmake**:

DXF names of entities introduced prior to AutoCAD Release 13	
3DFACE	ARC
ATTDEF	ATTRIB
CIRCLE	DIMENSION
INSERT	LINE
POINT	POLYLINE (old-style)
SEQEND	SHAPE
SOLID	TEXT
TRACE	VERTEX
VIEWPORT	

The **entmake** function verifies that a valid layer name, linetype name, and color are supplied. If a new layer name is introduced, **entmake** automatically creates the new layer. The **entmake** function also checks for block names, dimension style names, text style names, and shape names, if the entity type requires them. The function fails if it cannot create valid entities. Objects created on a frozen layer are not regenerated until the layer is thawed.

Creating Complex Entities

To create a complex entity (an old-style polyline or a block), you make multiple calls to **entmake**, 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 present. (See Extended Data—xdata (page 106) .) For each subsequent **entmake** call, the

function checks 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 (that is, when **entmake** receives an appropriate seqend or endblk subentity), the entity is checked for consistency; if valid, it is added to the drawing. The file is deleted when the complex entity is complete or when its creation has been canceled.

No portion of a complex entity is displayed on your drawing until its definition is complete. The entity does not appear in the drawing database until the final seqend or endblk subentity has been passed to **entmake** . The **entlast** function cannot retrieve the most recently created subentity for a complex entity that has not been completed. You can cancel the creation of a complex entity by entering **entmake** with no arguments. This clears the temporary file and returns `nil`.

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, an old-style polyline. The polyline has a linetype of DASHED and a color 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")    ; Object type
           (62 . 5)            ; Color
           (6 . "dashed")      ; Linetype
           (66 . 1)            ; Vertices follow
) )
(entmake '((0 . "VERTEX")      ; Object type
           (10 1.0 1.0 0.0)    ; Start point
) )
(entmake '((0 . "VERTEX")      ; Object type
           (10 4.0 6.0 0.0)    ; Second point
) )
(entmake '((0 . "VERTEX")      ; Object type
           (10 3.0 2.0 0.0)    ; Third point
) )
(entmake '((0 . "SEQEND")))    ; Sequence end
```

When defining dotted pairs, as in the above example, there must be a space on both sides of the dot. Otherwise, you will get an invalid dotted pair error message.

Block definitions begin with a block entity and end with an endblk subentity. Newly created blocks are automatically entered into the symbol table where they can be referenced. Block definitions cannot be nested, nor can they reference themselves. A block definition can contain references to other block definitions.

NOTE Before you use **entmake** to create a block, you should use **tblsearch** to ensure that the name of the new block is unique. The **entmake** function does not check for name conflicts in the block definitions table, so it can redefine existing blocks. See Symbol Table and Dictionary Access (page 114) for information on using **tblsearch**.

Block references can include an attributes-follow flag (group 66). If present and equal to 1, a series of attribute (attrib) entities is expected to follow the insert object. The attribute sequence is terminated by a seqend subentity.

Old-style polyline entities always include a vertices-follow flag (also group 66). The value of this flag must be 1, and the flag must be followed by a sequence of vertex entities, terminated by a seqend subentity.

Applications can 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 dividing them into triangular wedges. AutoCAD represents triangular wedges as four-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.

Complex entities can exist in either model space or paper space, but not both. If you have changed the current space by invoking either MSPACE or PSPACE (with **command**) while a complex entity is being constructed, a subsequent call to **entmake** cancels 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.

Working with Blocks

There is no direct method for an application to check whether a block listed in the BLOCK table is actually referenced by an insert object in the drawing. You can use the following code to scan the drawing for instances of a block reference:

```
(ssget "x" '((2 . "BLOCKNAME")))
```

You must also scan each block definition for instances of nested blocks.

Anonymous Blocks

The block definitions (BLOCK) table in a drawing can contain anonymous blocks (also known as unnamed blocks), that AutoCAD creates to support hatch patterns and associative dimensioning. The **entmake** function can create anonymous blocks other than `*D`*nnn* (dimensions) and `*X`*nnn* (hatch patterns). Unreferenced anonymous blocks are purged from the BLOCK definition table when a drawing is opened. Referenced anonymous blocks (those that have been inserted) are *not* purged. You can use **entmake** to create a block reference (insert object) to an anonymous block. (You *cannot* pass an anonymous block to the INSERT command.) Also, you can use **entmake** to redefine the block. You can modify the entities in a block (but not the block object itself) with **entmod**.

The name (group 2) of an anonymous block created by AutoLISP or ObjectARX 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 1. 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.

NOTE Anonymous block names do not remain constant. Although a referenced anonymous block becomes permanent, the numeric portion of its name can change between drawing sessions.

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: When **entmod** modifies a subentity, it does not update the image of the entire (complex) entity. If, for example, an application modifies 100 vertices of an old-style polyline with 100 calls to **entmod**, the time required to recalculate and redisplay the entire polyline is unacceptably slow. Instead, an application can perform a series of subentity modifications, and then redisplay the entire entity with a single call to the **entupd** function.

Consider the following: If the first entity in the current drawing is an old-style polyline with several vertices, the following code modifies the second vertex of the polyline and 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, **entupd** can regenerate any entity in the current drawing.

NOTE To ensure that all instances of the block references are updated, you must regenerate the drawing by invoking the AutoCAD REGEN command (with **command**). The **entupd** function is not sufficient if the modified entity is in a block definition.

Old-Style Polylines and Lightweight Polylines

A lightweight polyline (lwpolyline) is defined in the drawing database as a single graphic entity. The lwpolyline differs from the old-style polyline, which is defined as a group of subentities. Lwpolylines display faster and consume less disk space and RAM.

As of Release 14 of AutoCAD, 3D polylines are always created as old-style polyline entities, and 2D polylines are created as lwpolyline entities, unless they are curved or fitted with the PEDIT command. When a drawing from an earlier release is opened in Release 14 or a later release, all 2D polylines convert to lwpolylines automatically, unless they have been curved or fitted or contain xdata.

Processing Curve-Fit and Spline-Fit Polylines

When an AutoLISP application uses **entnext** to step through the vertices of an old-style polyline, it might encounter vertices that were not created explicitly. Auxiliary vertices are inserted automatically by the PEDIT command's Fit and Spline options. You can safely ignore them, because changes to these vertices will be discarded the next time the user applies PEDIT to fit or to spline the polyline.

The old-style 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 bit is set, all the polyline's vertices are regular user-defined vertices. However, if the curve-fit bit (2) is set, alternating vertices of the polyline have the bit value 1 set in their 70 group to indicate that they were inserted by the curve-fitting process. If you use **entmod** to move the vertices of such a polyline with the intent of refitting the curve by means of PEDIT, ignore these vertices.

Likewise, if the old-style 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 use **entmod** to move the vertices and you intend to refit the spline afterward, move only the control-point vertices.

Non-Graphic Object Handling

AutoCAD uses two types of non-graphical objects: dictionary objects and symbol table objects. Although there are similarities between these object types, they are handled differently.

All object types are supported by the **entget**, **entmod**, **entdel**, and **entmake** functions, although object types individually dictate their participation in these functions and may refuse any or all processing. With respect to AutoCAD built-in objects, the rules apply for symbol tables and for dictionary objects. For more information, see Symbol Table Objects (page 104) and Dictionary Objects (page 106).

All rules and restrictions that apply to graphic objects apply to non-graphic objects as well. Non-graphic objects cannot be passed to the **entupd** function.

When using **entmake**, the object type determines where the object will reside. For example, if a layer object is passed to **entmake**, it automatically goes to the layer symbol table. If a graphic object is passed to **entmake**, it will reside in the current space (model or paper).

Symbol Table Objects

The following rules apply to symbol tables:

- Symbol table entries can be created through **entmake** with few restrictions, other than being valid record representations, and name conflicts can only occur in the VPORT table. *ACTIVE entries cannot be created.
- Symbol table entries cannot be deleted with **entdel**.
- The object states of symbol tables and symbol table entries may be accessed with **entget** by passing the entity name. The **tblobjname** function can be used to retrieve the entity name of a symbol table entry.
- Symbol tables themselves cannot be created with **entmake**; however, symbol table entries can be created with **entmake**.
- Handle groups (5, 105) cannot be changed in **entmod**, nor specified in **entmake**.
- Symbol table entries that are not in the APPID table can have many of their fields modified with **entmod**. To be passed to **entmod**, a symbol table record list must include its entity name, which can be obtained from **entget**

but not from the **tblsearch** and **tblnext** functions. The 70 group of symbol table entries is ignored in **entmod** and **entmake** operations.

Renaming symbol table entries to duplicate names is not acceptable, except for the VPORT symbol table. The following entries cannot be modified or renamed, except that most LAYER entries can be renamed and xdata can be modified on all symbol table entries.

Symbol table entries that cannot be modified or renamed

Table	Entry name
VPORT	*ACTIVE
LINETYPE	CONTINUOUS
LAYER	Entries cannot be modified, except for xdata, but renaming is allowed

The following entries cannot be renamed, but are otherwise modifiable:

Symbol table entries that cannot be renamed

Table	Entry name
STYLE	STANDARD
DIMSTYLE	STANDARD
BLOCKS	*MODEL_SPACE
BLOCKS	*PAPER_SPACE
APPID	No entries can be renamed

Dictionary Objects

The following rules apply to dictionary objects:

- Dictionary objects can be examined with **entget** and their xdata modified with **entmod**. Their entries cannot be altered with **entmod**. All access to their entries are made through the **dictsearch** and **dictnext** functions.
- Dictionary entry contents cannot be modified through **entmod**, although xdata can be modified.
- Dictionary entries that begin with ACAD* cannot be renamed.

Extended Data - xdata

Several AutoLISP functions are provided to handle extended data (xdata), which is created by applications written with ObjectARX or AutoLISP. If an entity contains xdata, it follows the entity's regular definition data.

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

When xdata is retrieved with **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 table and as described in the topics in this section.

Group codes for regular and extended data

Group code	Field	Type of data
(-1, -2 (0-239)	Entity name) Regular definition data fields) . . .	Normal entity definition data
(-3 (1001 (1000, 1002-1071	Extended data sentinel Registered application name 1) XDATA fields) .	Extended data

Group codes for regular and extended data

Group code	Field	Type of data
(1001 (1000, 1002-1071 (1001	. . Registered application name 2) XDATA fields) . . . Registered application name 3) . .	

Organization of Extended Data

Extended data consists of one or more 1001 groups, each of which begins with a unique application name. The xdata groups returned by **entget** follow the definition data in the order in which they are saved in the database.

Within each application's group, the contents, meaning, and organization of the data are defined by the application. AutoCAD maintains the information but does not use it. The table also shows that the group codes for xdata are in the range 1000-1071. Many of these group codes are for familiar data types, as follows:

String 1000. Strings in extended 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 32nd 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.

Layer Name 1003. Name of a layer associated with the xdata.

Database -Handle 1005. Handle of an entity in the drawing database.

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 use **entget** to retrieve an entity's definition list that contains a 1071 group, the value is returned as a real, as shown in the following example:

(1071 . 12.0)

If you want to create a 1071 group in an entity with **entmake** or **entmod**, you can use either a real or an integer value, as shown in the following 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) .... )))
```

The preceding statement returns the following:

```
(1071 . 65537.0)
```

ObjectARX always manages 1071 groups as long integers.

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

Control String 1002. An xdata 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 the right brace terminates the most recent list. Lists can be nested.

NOTE

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

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

NOTE 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 WCS coordinates are moved, scaled, rotated, and mirrored along with the parent entity to which

the extended data belongs. The WCS position is also stretched when the STRETCH command is applied to the parent entity and when 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. A 3D point that is rotated or mirrored along with the parent, but not scaled, stretched, or moved. The WCS 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 xdata are also described in the *DXF Reference*.

Registration of an Application

To be recognized by AutoCAD, an application must register the name or names that it uses. Application names are saved with the extended data of each entity that uses them, and also in the APPID table. Registration is done with the **regapp** function, which specifies a string to use as an application name. If it successfully adds the name to APPID, it returns the name of the application; otherwise it returns `nil`. A result of `nil` indicates that the name is already present in the symbol table. This is not an actual error condition but an expected return value, because the application name needs to be registered only once per drawing.

To register itself, an application should first check that its name is not already in the APPID table. If the name is not there, the application must register it. Otherwise, it can simply go ahead and use the data, as described later in this section.

The following fragment shows the typical use of **regapp**. (The **tblsearch** function is described in Symbol Table and Dictionary Access (page 114).)

```
(setq appname "MYAPP_2356")        ; Unique application
name.
(if  (tblsearch "appid" appname)   ; Checks if already
registered.
  (princ (strcat
    "\n" appname " already registered. "))
  (if (=  (regapp appname) nil)    ; Some other problem.
    (princ (strcat
```

```
        "\nCan't register XDATA for " appname ". "))
    )
  )
```

The **regapp** function provides a measure of security, but it cannot guarantee that two separate 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 and time).

Retrieval of Extended Data

An application can call **entget** to obtain the xdata that it has registered. The **entget** function can return both the definition data and the xdata 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 registered by a previous call to **regapp**; they can also contain wild-card characters.

By default, associative hatch patterns contain extended data. The following code shows the association list of this xdata.

```
Command: (entget (car (entsel)) '("ACAD"))
Select object: Select an associative hatch
```

Entering the preceding code at the command line returns 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 xdata for two specified applications. Note that the *application* argument passes application names in list form:

```
(setq working_elist
  (entget ent_name
    '("MY_APP_1" "SOME_OTHER")  ; Only xdata from "MY_APP_1"
```

```
  )                                ; and "SOME_OTHER" is
retrieved.
)

(if working_elist
  (progn
    ...                            ; Updates working entity
groups.
    (entmod working_elist)      ; Only xdata from registered

  )                                ; applications still in
the
)                                  ; working_elist list are
modified.
```

As the sample code shows, you can modify xdata retrieved by **entget** by using a subsequent call to **entmod**, just as you can use **entmod** to modify normal definition data. You can also create xdata by defining it in the entity list passed to **entmake**.

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

NOTE Because the strings passed by *application* can include wild-card characters, an application name of `"*"` will cause **entget** to return all extended data attached to an entity.

Attachment of Extended Data to an Entity

You can use xdata to store any type of information you want. For example, draw an entity (such as a line or a circle), then enter the following code to attach xdata to the entity:

```
(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
 '((-3 ("NEWDATA"                     ; exdata equal to the
  (1000 . "This is a new thing!") ; new extended data—
  )))                                 ; in this case, a text
)                                     ; string.
(setq newent
  (append lastent exdata))  ; Appends new data list to
                            ; entity's list.
(entmod newent)             ; Modifies the entity with the
 new
                            ; definition data.
```

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

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

This example shows the basic method for attaching extended data to an entity.

Management of Extended Data Memory Use

Extended data is currently limited to 16K per entity. Because the xdata of an entity can be created and maintained by multiple applications, problems can result when the size of the xdata approaches its limit. AutoLISP provides two functions, **xdsize** and **xdroom**, to assist in managing the memory that xdata occupies. When **xdsize** is passed a list of xdata, it returns the amount of memory (in bytes) that the 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.

The **xdsize** function reads an extended data list, which can be large. This function can be slow, so 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 didn't have enough room for the xdata.

Handles in Extended Data

Extended 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 xdata. The handle can be retrieved later from xdata and then

passed to **handent** to obtain the other entity. Because more than one entity can reference another, xdata 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 that xdata entity handles are valid is to obtain a referenced entity's handle directly from its definition data by means of **entget**. The handle value is in group 5.

When you reference entities in other drawings (for example, entities that are attached with XREF), you can avoid protests from AUDIT by using extended entity strings (group 1000) rather than handles (group 1005). The handles of cross-referenced entities are either not valid in the current drawing, or they conflict with valid handles. However, if an XREF Attach changes to an XREF Bind or is combined with the current drawing in some other way, it is up to the application to revise the entity references accordingly.

When drawings are combined by means of INSERT, INSERT*, XREF Bind (XBIND), or partial DXFIN, handles are translated so they become valid in the current drawing. (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 (with the BLOCK command), the entity within the block is assigned new handles. (If the original entity is restored by means of OOPS, it retains its original handles.) The value of any xdata handles remains unchanged. When a block is exploded (with the EXPLODE command), xdata handles are translated in a manner similar to the way they are translated when drawings are combined. If the xdata handle refers to an entity that is not within the block, it is unchanged. However, if the xdata handle refers to an entity that is within the block, the data handle is assigned the value of the new (exploded) entity's handle.

Xrecord Objects

Xrecord objects are used to store and manage arbitrary data. They are composed of DXF group codes with normal object groups (that is, non-xdata group codes), ranging from 1 through 369 for supported ranges. These objects are similar in concept to xdata but is not limited by size or order.

The following examples provide methods for creating and listing xrecord data.

```
(defun C:MAKEXRECORD( / xrec xname )
  ; create the xrecord's data list.
  (setq xrec '((0 . "XRECORD")(100 . "AcDbXrecord")
    (1 . "This is a test xrecord list")
```

```
      (10 1.0 2.0 0.0) (40 . 3.14159) (50 . 3.14159)
      (62 . 1) (70 . 180))
    )
    ; use entmakex to create the xrecord with no owner.
    (setq xname (entmakex xrec))
    ; add the new xrecord to the named object dictionary.
    (dictadd (namedobjdict) "XRECLIST" xname)
    (princ)
)

(defun C:LISTXRECORD ( / xlist )
  ; find the xrecord in the named object dictionary.
  (setq xlist (dictsearch (namedobjdict) "XRECLIST"))
  ; print out the xrecord's data list.
  (princ xlist)
  (princ)
)
```

Symbol Table and Dictionary Access

AutoLISP provides functions for accessing symbol table and dictionary entries. Examples of the **tblnext** and **tblsearch** functions are provided in the following sections. For a complete list of the symbol table and dictionary access functions, see Symbol Table and Dictionary-Handling Functions (page 146) in AutoLISP Function Synopsis, (page 119) Refer to the *AutoLISP Reference* for more detailed information on the functions listed in the Synopsis.

For additional information on non-graphic objects see, Non-Graphic Object Handling (page 104).

Symbol Tables

Symbol table entries can also be manipulated by the following functions:

- **entdel**
- **entget**
- **entmake**
- **entmod**
- **handent**

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, DIMSTYLE, and APPID. Both functions return lists with DXF group codes that are similar to 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.

In the following example, the function **GETBLOCK** retrieves the symbol table entry for the first block (if any) in the current drawing, and 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 new line, then

                                 ; each list member.
    (setq ct (1+ ct))            ; Increments the counter
 by 1.
  )
  (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 occupied blocks. In a drawing with a single block named BOX, a call to **GETBLOCK** displays the following. (The name value varies 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 you can use 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 useful when you are handling the `VPORT` symbol table, because all viewports in a particular viewport configuration have the same name (such as *ACTIVE).

If the `VPORT` symbol table is accessed when TILEMODE is turned off, any changes have no visible effect until TILEMODE is turned on. Do not confuse `VPORTS`,which is described by the `VPORT` symbol table with paper space viewport entities.

The following processes all viewports in the `4VIEW` configuration:

```
(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.
)
```

Dictionary Entries

A dictionary is a container object, similar to the symbol tables in functions. Dictionary entries can be queried with the **dictsearch** and **dictnext** functions. Each dictionary entry consists of a text name key plus a hard ownership handle reference to the entry object. Dictionary entries may be removed by directly passing entry object names to the **entdel** function. The text name key uses the same syntax and valid characters as symbol table names.

Accessing AutoCAD Groups

The following is an example of one method for accessing the entities contained in a group. This example assumes a group named G1 exists in the current drawing.

```
(setq objdict (namedobjdict))
(setq grpdict (dictsearch objdict "ACAD_GROUP"))
```

This sets the `grpdict` variable to the entity definition list of the ACAD_GROUP dictionary and returns the following:

```
((-1 . <Entity name: 8dc10468>) (0 . "DICTIONARY") (5 . "D")
(102 . "{ACAD_REACTORS") (330 . <Entity name: 8dc10460>)
(102 . "}") (100 . "AcDbDictionary") (3 . "G1")
(350 . <Entity name: 8dc41240>))
```

The following code sets the variable `group1` to the entity definition list of the G1 group:

```
(setq group1 (dictsearch (cdar grpdict) "G1"))
```

It returns the following:

```
((-1 . <Entity name: 8dc10518>) (0 . "GROUP") (5 . "23")
(102 . "{ACAD_REACTORS") (330 . <Entity name: 8dc10468>)
(102 . "}") (100 . "AcDbGroup") (300 . "line and circle")
(70 . 0) (71 . 1)
(340 . <Entity name: 8dc10510>)(340 . <Entity name: 8dc10550>)
)
```

The 340 group codes are the entities that belong to the group.

Appendixes

3

AutoLISP Function Synopsis

To find a function without knowing its name, use the listings in this appendix. The AutoLISP® functions in this synopsis are organized into functional groups, and listed alphabetically within each function group. Each function is briefly described by its signature and a single sentence indicating the function's purpose.

Category Summary

Functions in this synopsis are organized into the following categories:

- *Basic:* Application-handling, arithmetic, equality and conditional, error-handling, function-handling, list manipulation, string-handling, and symbol-handling functions
- *Utility:* Conversion, device access, display control, file-handling, geometric, query and command, and user input functions
- *Selection Set, Object, and Symbol Table:* Extended data-handling, object-handling, selection set manipulation, and symbol table-handling functions
- *Memory Management*
- *VLX Namespace:* Function exposure, document namespace variable access, and error-handling functions
- *Namespace Communication:* Blackboard-addressing and multi-document loading functions
- *Property List (PIist) Handling*

Functions are grouped by data type and by the action they perform. Detailed information on each function is provided in the alphabetical listings in the *AutoLISP Reference*.

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

Basic Functions

Utility Functions

Selection Set, Object, and Symbol Table Functions

Basic Functions

The basic functions consist of the arithmetic, string-handling, equality and conditional, list manipulation, symbol-handling, function-handling, error-handling, and application-handling functions.

Application-Handling Functions

The following table provides summary descriptions of the AutoLISP application-handling functions.

Application-handling functions

Function	Description
(arx)	Returns a list of the currently loaded ObjectARX applications

Application-handling functions	
Function	**Description**
(arxload *application [onfailure]*)	Loads an ObjectARX application
(arxunload *application [onfailure]*)	Unloads an ObjectARX application
(autoarxload *filename cmdlist*)	Predefines command names to load an associated ObjectARX file
(autoload *filename cmdlist*)	Predefines command names to load an associated AutoLISP file
(initdia *[dialogflag]*)	Forces the display of the next command's dialog box
(load *filename [onfailure]*)	Evaluates the AutoLISP expressions in a file
(startapp *appcmd file*)	Starts an application
(vl-load-all *filename*)	Loads a file into all open AutoCAD documents

Arithmetic Functions

The following table provides summary descriptions of the AutoLISP arithmetic functions.

Arithmetic functions	
Function	**Description**
(+ (add) [*number number*] ...**)**	Returns the sum of all numbers
(- (subtract) *[number number]* ...)	Subtracts the second and following numbers from the first and returns the difference

Arithmetic functions

Function	Description
(* (multiply) *[number number]* ...)	Returns the product of all numbers
(/ (divide) *[number number]* ...)	Divides the first number by the product of the remaining numbers and returns the quotient
(~ (bitwise NOT) *int*)	Returns the bitwise NOT (1's complement) of the argument
(1+ (increment) *number*)	Returns the argument increased by 1 (incremented)
(1- (decrement) *number*)	Returns the argument reduced by 1 (decremented)
(abs *number*)	Returns the absolute value of the argument
(atan *num1 [num2]*)	Returns the arctangent of a number in radians
(cos *ang*)	Returns the cosine of an angle expressed in radians
(exp *number*)	Returns the constant e (a real) raised to a specified power (the natural antilog)
(expt *base power*)	Returns a number raised to a specified power
(fix *number*)	Returns the conversion of a real into the nearest smaller integer
(float *number*)	Returns the conversion of a number into a real
(gcd *int1 int2*)	Returns the greatest common denominator of two integers

Arithmetic functions	
Function	**Description**
(log *number*)	Returns the natural log of a number as a real
(logand [*int int* ...])	Returns the result of the logical bitwise AND of a list of integers
(logior [*int int* ...])	Returns the result of the logical bitwise inclusive OR of a list of integers
(lsh [*int numbits*])	Returns the logical bitwise shift of an integer by a specified number of bits
(max [*number number* ...])	Returns the largest of the numbers given
(min [*number number* ...])	Returns the smallest of the numbers given
(minusp *number*)	Verifies that a number is negative
(rem [*num1 num2* ...])	Divides the first number by the second, and returns the remainder
(sin *ang*)	Returns the sine of an angle as a real expressed in radians
(sqrt *number*)	Returns the square root of a number as a real
(zerop *number*)	Verifies that a number evaluates to zero

Equality and Conditional Functions

The following table provides summary descriptions of the AutoLISP equality and conditional functions.

Equality and conditional functions

Function	Description
(= (equal to) *numstr [numstr]* ...)	Returns T if all arguments are numerically equal, and returns nil otherwise
(/= (not equal to) *numstr [numstr]* ...)	Returns T if the arguments are not numerically equal, and nil if the arguments are numerically equal
(< (less than) *numstr [numstr]* ...)	Returns T if each argument is numerically less than the argument to its right, and returns nil otherwise
(<= (less than or equal to) *numstr [numstr]* ...)	Returns T if each argument is numerically less than or equal to the argument to its right, and returns nil otherwise
(> (greater than) *numstr [numstr]* ...)	Returns T if each argument is numerically greater than the argument to its right, and returns nil otherwise
(>= (greater than or equal to) *numstr [numstr]* ...)	Returns T if each argument is numerically greater than or equal to the argument to its right, and returns nil otherwise
(and [*expr* ...])	Returns the logical AND of a list of expressions
(Boole *func int1* [*int2* ...])	Serves as a general bitwise Boolean function
(cond [(*test result* ...) ...])	Serves as the primary conditional function for AutoLISP

Equality and conditional functions	
Function	**Description**
(eq *expr1 expr2*)	Determines whether two expressions are identical
(equal *expr1 expr2 [fuzz]*)	Determines whether two expressions are equal
(if *testexpr thenexpr [elseexpr]*)	Conditionally evaluates expressions
(or [*expr* ...])	Returns the logical OR of a list of expressions
(repeat *int* [*expr* ...])	Evaluates each expression a specified number of times, and returns the value of the last expression
(while *testexpr* [*expr* ...])	Evaluates a test expression, and if it is not nil, evaluates other expressions; repeats this process until the test expression evaluates to nil

Error-Handling Functions

The following table provides summary descriptions of the AutoLISP error-handling functions.

Error-handling functions	
Function	**Description**
(alert *string*)	Displays an alert dialog box with the error or warning message passed as a string
(*error* *string*)	A user-definable error-handling function
(exit)	Forces the current application to quit
(quit)	Forces the current application to quit

Error-handling functions	
Function	**Description**
(vl-catch-all-apply ' *function list*)	Passes a list of arguments to a specified function and traps any exceptions
(vl-catch-all-error-message *error-obj*)	Returns a string from an error object
(vl-catch-all-error-p *arg*)	Determines whether an argument is an error object returned from **vl-catch-all-apply**

Function-Handling Functions

The following table provides summary descriptions of the AutoLISP function-handling functions.

Function-handling functions	
Function	**Description**
(apply *function lst*)	Passes a list of arguments to a specified function
(defun *sym* ([*arguments*] [/ *variables*...]) *expr* ...)	Defines a function
(defun-q *sym* ([*arguments*] [/ *variables*...]) *expr* ...)	Defines a function as a list (intended for backward-compatibility only)
(defun-q-list-ref ' *function*)	Displays the list structure of a function defined with **defun-q**
(defun-q-list-set *'sym list*)	Defines a function as a list (intended for backward-compatibility only)
(eval *expr*)	Returns the result of evaluating an AutoLISP expression

Function-handling functions	
Function	**Description**
(lambda *arguments expr* ...)	Defines an anonymous function
(progn *[expr]* ...)	Evaluates each expression sequentially, and returns the value of the last expression
(trace *function* ...)	Aids in AutoLISP debugging
(untrace *function* ...)	Clears the trace flag for the specified functions

List Manipulation Functions

The following table provides summary descriptions of the AutoLISP list manipulation functions.

List manipulation functions	
Function	**Description**
(acad_strlsort *lst*)	Sorts a list of strings by alphabetical order
(append *lst* ...)	Takes any number of lists and runs them together as one list
(assoc *item alist*)	Searches an association list for an element and returns that association list entry
(car *lst*)	Returns the first element of a list
(cdr *lst*)	Returns the specified list, except for the first element of the list
(cons *new-first-element lst*)	The basic list constructor

List manipulation functions

Function	Description
(foreach *name lst* [*expr* ...])	Evaluates expressions for all members of a list
(last *lst*)	Returns the last element in a list
(length *lst*)	Returns an integer indicating the number of elements in a list
(list [*expr* ...])	Takes any number of expressions and combines them into one list
(listp *item*)	Verifies that an item is a list
(mapcar *function list1 ... listn*)	Returns a list of the result of executing a function with the individual elements of a list or lists supplied as arguments to the function
(member *expr lst*)	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 lst*)	Returns the nth element of a list
(reverse *lst*)	Returns a list with its elements reversed
(subst *newitem olditem lst*)	Searches a list for an old item and returns a copy of the list with a new item substituted in place of every occurrence of the old item
(vl-consp *list-variable*)	Determines whether or not a list is nil

List manipulation functions	
Function	**Description**
(vl-every *predicate-function list* [*more-lists*]...)	Checks whether the predicate is true for every element combination
(vl-list* *object* [*more-objects*]...)	Constructs and returns a list
(vl-list->string *char-codes-list*)	Combines the characters associated with a list of integers into a string
(vl-list-length *list-or-cons-object*)	Calculates list length of a true list
(vl-member-if *predicate-function list*)	Determines whether the predicate is true for one of the list members
(vl-member-if-not *predicate-function list*)	Determines whether the predicate is nil for one of the list members
(vl-position *symbol list*)	Returns the index of the specified list item
(vl-remove *element-to-remove list*)	Removes elements from a list
(vl-remove-if *predicate-function list*)	Returns all elements of the supplied list that fail the test function
(vl-remove-if-not *predicate-function list*)	Returns all elements of the supplied list that pass the test function
(vl-some *predicate-function list* [*more-lists*]...)	Checks whether the predicate is not nil for one element combination
(vl-sort *list* less?-function)	Sorts the elements in a list according to a given compare function

List manipulation functions	
Function	**Description**
(vl-sort-i *list* less?-function)	Sorts the elements in a list according to a given compare function, and returns the element index numbers
(vl-string->list *string*)	Converts a string into a list of character codes

String-Handling Functions

The following table provides summary descriptions of the AutoLISP string-handling functions.

String-handling functions	
Function	**Description**
(read *[string]*)	Returns the first list or atom obtained from a string
(strcase *string* [*which*])	Returns a string where all alphabetic characters have been converted to uppercase or lowercase
(strcat [string1 [string2] ...)	Returns a string that is the concatenation of multiple strings
(strlen [string] ...)	Returns an integer that is the number of characters in a string
(substr string start [length])	Returns a substring of a string
(vl-prin1-to-string *object*)	Returns the string representation of any LISP object as if it were output by the **prin1** function

String-handling functions

Function	Description
(vl-princ-to-string *object*)	Returns the string representation of any LISP object as if it were output by the **princ** function
(vl-string->list *string*)	Converts a string into a list of character codes
(vl-string-elt *string position*)	Returns the ASCII representation of the character at a specified position in a string
(vl-string-left-trim *character-set string*)	Removes the specified characters from the beginning of a string
(vl-string-mismatch *str1 str2* [*pos1 pos2 ignore-case-p*])	Returns the length of the longest common prefix for two strings, starting at specified positions
(vl-string-position *char-code str* [*start-pos* [*from-end-p*]])	Looks for a character with the specified ASCII code in a string
(vl-string-right-trim *character-set string*)	Removes the specified characters from the end of a string
(vl-string-search *pattern string* [*start-pos*])	Searches for the specified pattern in a string
(vl-string-subst *new-str pattern string* [*start-pos*])	Substitutes one string for another, within a string
(vl-string-translate *source-set dest-set str*)	Replaces characters in a string with a specified set of characters
(vl-string-trim *char-set str*)	Removes the specified characters from the beginning and end of a string

String-handling functions	
Function	**Description**
(wcmatch string pattern)	Performs a wild-card pattern match on a string

Symbol-Handling Functions

The following table provides summary descriptions of the AutoLISP symbol-handling functions.

Symbol-handling functions	
Function	**Description**
(atom*item*)	Verifies that an item is an atom
(atoms-family *format [symlist]*)	Returns a list of the currently defined symbols
(boundp *sym*)	Verifies whether a value is bound to a symbol
(not *item*)	Verifies that an item evaluates to nil
(null *item*)	Verifies that an item is bound to nil
(numberp *item*)	Verifies that an item is a real or an integer
(quote *expr*)	Returns an expression without evaluating it
(set *sym expr*)	Sets the value of a quoted symbol name to an expression
(setq *sym1 expr1 [sym2 expr2] ...*)	Sets the value of a symbol or symbols to associated expressions
(type *item*)	Returns the type of a specified item

Symbol-handling functions	
Function	**Description**
(vl-symbol-name *symbol*)	Returns a string containing the name of a symbol
(vl-symbol-value *symbol*)	Returns the current value bound to a symbol
(vl-symbolp *object*)	Identifies whether or not a specified object is a symbol

Utility Functions

The utility functions consist of query and command, display control, user input, geometric, conversion, file-handling, and device access functions.

Conversion Functions

The following table provides summary descriptions of the AutoLISP conversion functions.

Conversion functions	
Function	**Description**
(angtof *string [mode]*)	Converts a string representing an angle into a real (floating-point) value in radians
(angtos *angle [mode [precision]]*)	Converts an angular value in radians into a string
(ascii *string*)	Returns the conversion of the first character of a string into its ASCII character code (an integer)
(atof *string*)	Returns the conversion of a string into a real
(atoi *string*)	Returns the conversion of a string into an integer

Conversion functions	
Function	**Description**
(chr *integer*)	Returns the conversion of an integer representing an ASCII character code into a single-character string
(cvunit *value from to*)	Converts a value from one unit of measurement to another
(distof *string [mode]*)	Converts a string that represents a real (floating-point) value into a real value
(itoa *int*)	Returns the conversion of an integer into a string
(rtos *number [mode [precision]]*)	Converts a number into a string
(trans *pt from to [disp]*)	Translates a point (or a displacement) from one coordinate system to another

Device Access Functions

The following table provides summary descriptions of the AutoLISP device access functions.

Device access functions	
Function	**Description**
(grread *[track] [allkeys [curtype]]*)	Reads values from any of the AutoCAD input devices

Display Control Functions

The following table provides summary descriptions of the AutoLISP display control functions.

Display control functions

Function	Description
(graphscr)	Displays the AutoCAD graphics screen
(grdraw *from to color [highlight]*)	Draws a vector between two points, in the current viewport
(grtext *[box text [highlight]]*)	Writes text to the status line or to screen menu areas
(grvecs *vlist [trans]*)	Draws multiple vectors on the graphics screen
(prin1 *[expr [file-desc]]*)	Prints an expression to the command line or writes an expression to an open file
(princ *[expr [file-desc]]*)	Prints an expression to the command line, or writes an expression to an open file
(print *[expr [file-desc]]*)	Prints an expression to the command line, or writes an expression to an open file
(prompt *msg*)	Displays a string on your screen's prompt area
(redraw *[ename [mode]]*)	Redraws the current viewport or a specified object (entity) in the current viewport
(terpri)	Prints a newline to the Command line
(textpage)	Switches from the graphics screen to the text screen

Display control functions	
Function	**Description**
(textscr)	Switches from the graphics screen to the text screen
(vports)	Returns a list of viewport descriptors for the current viewport configuration

File-Handling Functions

The following table provides summary descriptions of the AutoLISP file-handling functions.

File-handling functions	
Function	**Description**
(close *file-desc*)	Closes an open file
(findfile *filename*)	Searches the AutoCAD library path for the specified file
(open *filename mode*)	Opens a file for access by the AutoLISP I/O functions
(read-char *[file-desc]*)	Returns the decimal ASCII code representing the character read from the keyboard input buffer or from an open file
(read-line *[file-desc]*)	Reads a string from the keyboard or from an open file
(vl-directory-files [*directory pattern directories*])	Lists all files in a given directory

File-handling functions

Function	Description
(vl-file-copy "*source-filename*" "*destination-filename*" [*append*?])	Copies or appends the contents of one file to another file
(vl-file-delete "*filename*")	Deletes a file
(vl-file-directory-p "*filename*")	Determines if a file name refers to a directory
(vl-file-rename "*old-filename*" "*new-filename*")	Renames a file
(vl-file-size "*filename*")	Determines the size of a file, in bytes
(vl-file-systime "*filename*")	Returns last modification time of the specified file
(vl-filename-base "*filename*")	Returns the name of a file, after stripping out the directory path and extension
(vl-filename-directory "*filename*")	Returns the directory path of a file, after stripping out the name and extension
(vl-filename-extension "*filename*")	Returns the extension from a file name, after stripping out the rest of the name
(vl-filename-mktemp ["pattern" "directory" "extension"])	Calculates a unique file name to be used for a temporary file
(write-char *num [file-desc]*)	Writes one character to the screen or to an open file
(write-line *string [file-desc]*)	Writes a string to the screen or to an open file

Geometric Functions

The following table provides summary descriptions of the AutoLISP geometric functions.

Geometric functions	
Function	**Description**
(angle *pt1 pt2*)	Returns an angle in radians of a line defined by two endpoints
(distance *pt1 pt2*)	Returns the 3D distance between two points
(inters *pt1 pt2 pt3 pt4 [onseg]*)	Finds the intersection of two lines
(osnap *pt mode*)	Returns a 3D point that is the result of applying an Object Snap mode to a specified point
(polar *pt ang dist*)	Returns the UCS 3D point at a specified angle and distance from a point
(textbox *elist*)	Measures a specified text object, and returns the diagonal coordinates of a box that encloses the text

Query and Command Functions

The following table provides summary descriptions of the AutoLISP query and command functions.

Query and command functions	
Function	**Description**
(command *[arguments]* ...)	Executes an AutoCAD command

Query and command functions	
Function	**Description**
(command-s *[arguments]* ...)	Executes an AutoCAD command and the supplied input
(getcfg *cfgname*)	Retrieves application data from the AppData section of the *acad.cfg* file
(getcname *cname*)	Retrieves the localized or English name of an AutoCAD command
(getenv "*variable-name*")	Returns the string value assigned to an environment variable
(getvar *varname*)	Retrieves the value of an AutoCAD system variable
(help *[helpfile [topic [command]]]*)	Invokes the Help facility
(setcfg *cfgname cfgval*)	Writes application data to the AppData section of the *acad.cfg* file
(setenv "*varname*" "*value*")	Sets an environment variable to a specified value
(setfunhelp "*c:fname*" *["helpfile" ["topic" ["command"]]]*)	Registers a user-defined command with the Help facility so the appropriate help file and topic are called when the user requests help on that command
(setvar *varname value*)	Sets an AutoCAD system variable to a specified value
(ver)	Returns a string that contains the current AutoLISP version number

Query and command functions	
Function	**Description**
(vl-cmdf *[arguments]* ...)	Executes an AutoCAD command after evaluating *arguments*

User Input Functions

The following table provides summary descriptions of the AutoLISP user input functions.

User input functions	
Function	**Description**
(entsel *[msg]*)	Prompts the user to select a single object (entity) by specifying a point
(getangle *[pt] [msg]*)	Pauses for user input of an angle, and returns that angle in radians
(getcorner *pt [msg]*)	Pauses for user input of a rectangle's second corner
(getdist *[pt] [msg]*)	Pauses for user input of a distance
(getfiled *title default ext flags*)	Prompts the user for a file name with the standard AutoCAD file dialog box, and returns that file name
(getint *[msg]*)	Pauses for user input of an integer, and returns that integer
(getkword *[msg]*)	Pauses for user input of a keyword, and returns that keyword
(getorient *[pt] [msg]*)	Pauses for user input of an angle, and returns that angle in radians

User input functions	
Function	**Description**
(getpoint *[pt] [msg]*)	Pauses for user input of a point, and returns that point
(getreal *[msg]*)	Pauses for user input of a real number, and returns that real number
(getstring *[cr] [msg]*)	Pauses for user input of a string, and returns that string
(initget *[bits] [string]*)	Establishes keywords for use by the next user input function call
(nentsel *[msg]*)	Prompts the user to select an object (entity) by specifying a point, and provides access to the definition data contained within a complex object
(nentselp *[msg] [pt]*)	Provides similar functionality to that of the nentsel function without the need for user input

Selection Set, Object, and Symbol Table Functions

The selection set, object, and symbol table functions consist of selection set manipulation, object-handling, extended data-handling, and symbol table-handling functions.

Extended Data-Handling Functions

The following table provides summary descriptions of the AutoLISP extended data-handling functions.

Extended data-handling functions

Function	Description
(regapp *application*)	Registers an application name with the current AutoCAD drawing in preparation for using extended object data
(xdroom *ename*)	Returns the amount of extended data (xdata) space that is available for an object (entity)
(xdsize *lst*)	Returns the size (in bytes) that a list occupies when it is linked to an object (entity) as extended data

Object-Handling Functions

The following table provides summary descriptions of the AutoLISP object-handling functions.

Object-handling functions

Function	Description
(dumpallproperties*ename [context]*)	Retrieves an entity's supported properties
(entdel *ename*)	Deletes objects (entities) or restores previously deleted objects
(entget *ename [applist]*)	Retrieves an object's definition data
(entlast)	Returns the name of the last nondeleted main object in the drawing

Object-handling functions	
Function	**Description**
(entmake *[elist]*)	Creates a new entity (graphical object) in the drawing
(entmakex *[elist]*)	Makes a new object, gives it a handle and entity name (but does not assign an owner), and then returns the new entity name
(entmod *elist*)	Modifies the definition data of an object
(entnext *[ename]*)	Returns the name of the next object in the drawing
(entupd *ename*)	Updates the screen image of an object
(getpropertyvalue*ename property-name [or collectionName index name]*)	Returns the current value of an entity's property
(handent *handle*)	Returns an object name based on its handle
(ispropertyreadonly*ename property-name [or collectionName index name]*)	Returns the read-only state of an entity's property
(setpropertyvalue*ename property-name value [or collectionname index name val]*)	Sets the property value for an entity

Selection Set Manipulation Functions

The following table provides summary descriptions of the AutoLISP selection set manipulation functions.

Selection set manipulation functions

Function	Description
(ssadd *[ename [ss]]*)	Adds an object (entity) to a selection set, or creates a new selection set
(ssdel *ename ss*)	Deletes an object (entity) from a selection set
(ssget *[mode] [pt1 [pt2]] [pt-list] [filter-list]*)	Prompts the user to select objects (entities), and returns a selection set
(ssgetfirst)	Determines which objects are selected and gripped
(sslength *ss*)	Returns an integer containing the number of objects (entities) in a selection set
(ssmemb *ename ss*)	Tests whether an object (entity) is a member of a selection set
(ssname *ss index*)	Returns the object (entity) name of the indexed element of a selection set
(ssnamex *ss index*)	Retrieves information about how a selection set was created
(sssetfirst *gripset [pickset]*)	Sets which objects are selected and gripped

Symbol Table and Dictionary-Handling Functions

The following table provides summary descriptions of the AutoLISP symbol table and dictionary-handling functions.

Symbol table and dictionary-handling functions

Function	Description
(dictadd *ename symbol newobj*)	Adds a non-graphical object to the specified dictionary
(dictnext *ename symbol [rewind]*)	Finds the next item in a dictionary
(dictremove *ename symbol*)	Removes an entry from the specified dictionary
(dictrename *ename oldsym newsym*)	Renames a dictionary entry
(dictsearch *ename symbol [setnext]*)	Searches a dictionary for an item
(layoutlist)	Returns a list of all paper space layouts in the current drawing
(namedobjdict)	Returns the entity name of the current drawing's named object dictionary, which is the root of all non-graphical objects in the drawing
(setview *view_description [vport_id]*)	Establishes a view for a specified viewport
(snvalid *sym_name*)	Checks the symbol table name for valid characters
(tblnext *table-name [rewind]*)	Finds the next item in a symbol table
(tblobjname *table-name symbol*)	Returns the entity name of a specified symbol table entry

Symbol table and dictionary-handling functions	
Function	**Description**
(tblsearch *table-name symbol [setnext]*)	Searches a symbol table for a symbol name

Memory Management Functions

The following table provides summary descriptions of the AutoLISP memory management functions.

Memory management functions	
Function	**Description**
(alloc *int*)	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, which frees up unused memory
(mem)	Displays the current state of memory in AutoLISP

VLX Namespace Functions

The VLX namespace functions listed below apply to separate-namespace VLX applications. These functions allow separate-namespace VLX functions to be accessible from a document namespace, enable the retrieval and updating of

variables in the associated document namespace, and provide error-handling routines for separate-namespace VLX functions.

VLX namespace functions

Function	Description
(vl-doc-ref *symbol*)	Retrieves the value of a variable from the namespace of the associated document
(vl-doc-set *symbol value*)	Sets the value of a variable in the associated document's namespace
(vl-exit-with-error "*msg*")	Passes control from a VLX error handler to the ***error*** function of the associated document namespace
(vl-exit-with-value *value*)	Returns a value to the document namespace from which the VLX was invoked

Namespace Communication Functions

The namespace communication functions consist of blackboard addressing and multi-document-loading functions.

Namespace communication functions

Function	Description
(vl-bb-ref ' *variable*)	Returns the value of a variable from the blackboard namespace
(vl-bb-set ' *variable value*)	Sets the value of a variable in the blackboard namespace
(vl-load-all "*filename*")	Loads a file into all open AutoCAD documents, and into any document subsequently opened during the current AutoCAD session

Namespace communication functions	
Function	**Description**
(vl-propagate ' *variable*)	Copies the value of a variable into all open AutoCAD documents, and into any document subsequently opened during the current AutoCAD session

Property List (Plist) Functions

Property List functions query and update the Property List for AutoCAD.

Property List functions	
Function	**Description**
(vl-registry-delete *reg-key* [*val-name*])	Deletes the specified key or value from the Property List
(vl-registry-descendents *reg-key* [*val-names*])	Returns a list of subkeys or value names for the specified Property key
(vl-registry-read *reg-key* [*val-name*])	Returns data stored in the Property List for the specified key/value pair
(vl-registry-write *reg-key* [*val-name val-data*])	Creates a key in the Property List

AutoLISP Error Codes

This appendix lists the AutoLISP® error codes.

Error Codes

The following table shows the values of error codes generated by AutoLISP. The ERRNO system variable 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")**.

The ERRNO system variable is not always cleared to zero. Unless it is inspected immediately after an AutoLISP function has reported an error, the error that its value indicates may be misleading. This variable is always cleared when starting or opening a drawing.

NOTE The possible values of ERRNO, and their meanings, are subject to change.

Online program error codes

Value	Meaning
0	No error
1	Invalid symbol table name
2	Invalid entity or selection set name
3	Exceeded maximum number of selection sets
4	Invalid selection set
5	Improper use of block definition
6	Improper use of xref
7	Object selection: pick failed
8	End of entity file
9	End of block definition file
10	Failed to find last entity
11	Illegal attempt to delete viewport object
12	Operation not allowed during PLINE

Online program error codes	
Value	**Meaning**
13	Invalid handle
14	Handles not enabled
15	Invalid arguments in coordinate transform request
16	Invalid space in coordinate transform request
17	Invalid use of deleted entity
18	Invalid table name
19	Invalid table function argument
20	Attempt to set a read-only variable
21	Zero value not allowed
22	Value out of range
23	Complex REGEN in progress
24	Attempt to change entity type
25	Bad layer name
26	Bad linetype name
27	Bad color name
28	Bad text style name
29	Bad shape name

Online program error codes	
Value	**Meaning**
30	Bad field for entity type
31	Attempt to modify deleted entity
32	Attempt to modify seqend subentity
33	Attempt to change handle
34	Attempt to modify viewport visibility
35	Entity on locked layer
36	Bad entity type
37	Bad polyline entity
38	Incomplete complex entity in block
39	Invalid block name field
40	Duplicate block flag fields
41	Duplicate block name fields
42	Bad normal vector
43	Missing block name
44	Missing block flags
45	Invalid anonymous block
46	Invalid block definition

Online program error codes	
Value	**Meaning**
47	Mandatory field missing
48	Unrecognized extended data (XDATA) type
49	Improper nesting of list in XDATA
50	Improper location of APPID field
51	Exceeded maximum XDATA size
52	Entity selection: null response
53	Duplicate APPID
54	Attempt to make or modify viewport entity
55	Attempt to make or modify an xref, xdef, or xdep
56	ssget filter: unexpected end of list
57	ssget filter: missing test operand
58	ssget filter: invalid opcode (-4) string
59	ssget filter: improper nesting or empty conditional clause
60	ssget filter: mismatched begin and end of conditional clause
61	ssget filter: wrong number of arguments in conditional clause (for NOT or XOR)
62	ssget filter: exceeded maximum nesting limit
63	ssget filter: invalid group code

Online program error codes	
Value	**Meaning**
64	ssget filter: invalid string test
65	ssget filter: invalid vector test
66	ssget filter: invalid real test
67	ssget filter: invalid integer test
68	Digitizer is not a tablet
69	Tablet is not calibrated
70	Invalid tablet arguments
71	ADS error: Unable to allocate new result buffer
72	ADS error: Null pointer detected
73	Cannot open executable file
74	Application is already loaded
75	Maximum number of applications already loaded
76	Unable to execute application
77	Incompatible version number
78	Unable to unload nested application
79	Application refused to unload
80	Application is not currently loaded

Online program error codes	
Value	**Meaning**
81	Not enough memory to load application
82	ADS error: Invalid transformation matrix
83	ADS error: Invalid symbol name
84	ADS error: Invalid symbol value
85	AutoLISP/ADS operation prohibited while a dialog box was displayed

Index

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X