

# *Bonus Chapter 4: Introducing AutoLISP*

---

## *In This Chapter*

- ✓ **Accessing the AutoLISP Development Environment**
- ✓ **Using the VLIDE**
- ✓ **Creating a basic program**
- ✓ **Getting information to and from the user**
- ✓ **Using the debug tools in the Visual LISP Editor**
- ✓ **Using ActiveX automation with AutoLISP**

**A**utoLISP in AutoCAD is a feature that is often left undiscovered by many users. This is partly due to the number of new users who just don't have the time to master all the features AutoCAD offers. AutoCAD is also a much more robust program now than it was during its earlier years, so not as many people write custom programs with AutoLISP as they once did. AutoLISP is now more of a niche feature that helps to improve workflow, instead of a necessary tool to create features that are not part of the program.

Utilities such as Revision Cloud and Justify Text were originally AutoLISP programs found in AutoCAD; a majority of the Express Tools that ship with AutoCAD were also developed with AutoLISP. For more information on Express Tools, see Book IX, Chapter 4. As I discuss in this chapter, AutoLISP is a powerful language that is relatively easy to master. After you start creating your own custom programs, you might wonder why you didn't start programming with AutoLISP sooner.

## *Accessing the AutoLISP Development Environment*

Over the years, the main development environment for creating AutoLISP programs has been Notepad. Notepad was the tool of choice simply because Autodesk didn't offer a development tool for working with AutoLISP files. This all changed when Autodesk purchased a program called Vital LISP. Vital LISP was developed by a third-party software company and it extended the capabilities of AutoLISP. Shortly after Autodesk purchased Vital LISP, they renamed the program to Visual LISP.

Visual LISP has a complete AutoLISP development environment (see Figure BC4-1) that includes debugging tools and the capability to color-code

syntax, and much more. The development environment is known as the VLIDE — Visual LISP Integrated Development Environment. The VLIDE is accessible directly from AutoCAD.

### Launching the Visual LISP Editor

To launch the Visual LISP Editor, use one of these methods:

- ◆ On the ribbon, click Tools tab⇨Applications panel⇨Visual LISP Editor.
- ◆ On the menu browser or menu bar, click Tools menu⇨AutoLISP⇨Visual LISP Editor.
- ◆ At the command line, type **VLIDE** or **VLISP** and press Enter.

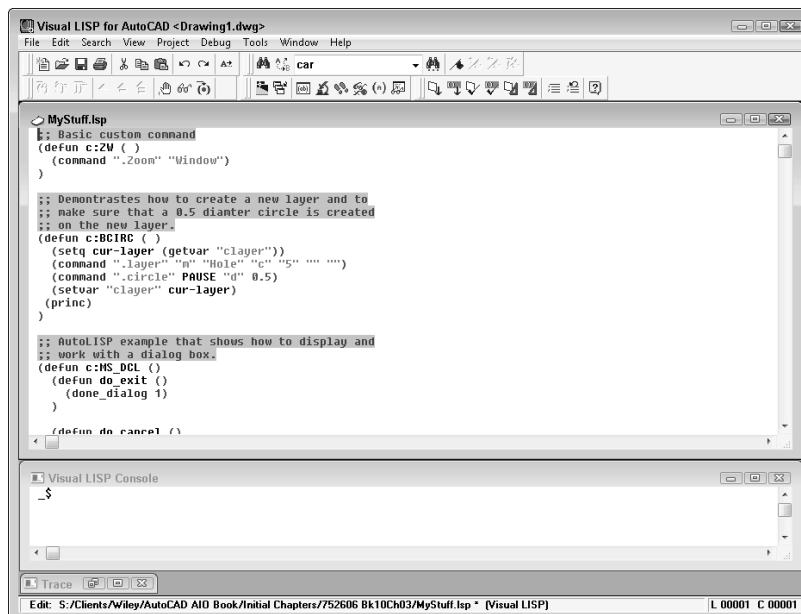
### Loading an existing AutoLISP application file

The following steps explain how to load an existing AutoLISP application file in the Visual LISP Editor:

1. On the menu browser or menu bar, click Tools menu⇨AutoLISP⇨Visual LISP Editor.

AutoCAD launches the Visual LISP Editor.

2. In the Visual LISP Editor, click Files⇨Open File.



**Figure BC4-1:**  
The Visual LISP Editor.

The Open File to Edit/View dialog box is displayed, allowing you to open AutoLISP, DCL, SQL, or C/C++ source files. In the Visual LISP Editor, you'll mainly be working with AutoLISP and DCL files.

**3. In the Files of Type drop-down list, specify the type of application file to use for filtering the list box.**

The selected option controls the type of application files that are displayed in the list box. The default selection is All Files. The drop-down list box allows you to filter on LSP, DCL, SQL, and C/C++ source files.

**4. In the Look In drop-down list, specify the location for the application file.**

The Look In drop-down list allows you to select a folder or a named location such as My Documents to start in.

**a. If the application file is located in a folder below the location that you selected in the Look In drop-down list, double-click the folder in the list box below the Look In drop-down list.**

The list box displays any folders or files located under the folder selected in the Look In drop-down list. The types of files displayed in the list box depend on the file filter selected in the Files of Type drop-down list.

**b. Keep double-clicking folders until you navigate to the folder that contains the application file you want to load. Select the application file from the list box.**

The application file is highlighted in the list box and its name is added to the File Name text box.

**5. Click Open.**

The application file opens in its own text window in the Visual LISP Editor.



If you are opening an AutoLISP file only for viewing, you can select the Open As Read-Only check box at the bottom of the Open File to Edit/View dialog box to avoid accidentally making changes to the file.

## Using the VLIDE

Although AutoLISP is not a mainstream programming language, it's nice to have a development environment dedicated to AutoLISP. To take full advantage of the development environment, you should understand some of its basic functionality and behavior.

The text window is where you edit and view source code that makes up an AutoLISP application file. This window allows you not only to edit your source code but also to run validation tools on your code to test it for things such as missing parentheses. Like most development environments, the main window controls how code is represented on-screen. Color-coding (see Figure BC4-2) is one of the biggest advantages that the Visual LISP editor has over an application such as Notepad. (Although this is a black-and-white book, the different colors are represented by different grayscale values in the figure.)

**Figure BC4-2:**  
AutoLISP said, "Let there be color!" and there was.

```

;; Demonstrates how to create a new layer and to
;; make sure that a 0.5 diameter circle is created
;; on the new layer
(defun c:BCIRC ( )
  (setq cur-layer (getvar "clayer"))
  (command ".layer" "m" "Hole" "c" "5" "" "" )
  (command ".circle" PAUSE "d" 0.5)
  (setvar "clayer" cur-layer)
  (princ)
)

```

Table BC4-1 lists some of the default colors for many of the common elements as they appear in the text window.

<b>Table BC4-1 Color-Coding the Text Window</b>		
<i>Window Element</i>	<i>Description</i>	<i>Default Color Scheme</i>
:WINDOW-TEXT	Variables and function names that are not protected	Black text with a white background
:LEX-SPACE	Empty space in the file that is represented by a space or tab	Black text with a white background
:LEX-STR	A string value that is surrounded with double quote marks	Magenta text with a white background
:LEX-SYM	Variables and function names that are protected	Blue text with a white background
:LEX-NUM	Whole number	Dark green text with a white background
:LEX-INT	Integer number	Dark green text with a white background
:LEX-REAL	Real or float number	Teal text with a white background
:LEX-PAREN	Parentheses	Red text with a white background
:LEX-COMM	Comment	Purple text with a gray background

## Controlling color-coding in the text window

You can change the default colors used for color-coding the text in the text window, but the color options are limited to a predetermined color palette. Follow these steps to change the color-coding scheme:

1. In the Visual LISP editor, click **Tools**→**Window Attributes**→**Configure Current**.

The Window Attributes dialog box (see Figure BC4-3) is displayed.

2. In the **Text Colors** section, click the drop-down list and select a window element from the list.

The drop-down list controls the window element that is currently being edited when the color swatches are clicked.

3. Click a color swatch in the top row to adjust the color to use for the text of the selected element.

The top row of color swatches controls the foreground color or the color of the text. The changes in the dialog box appear in both the dialog box along the top of the Text Colors section and in real time in the editor window, giving you a good understanding of how the changes affect the text window.

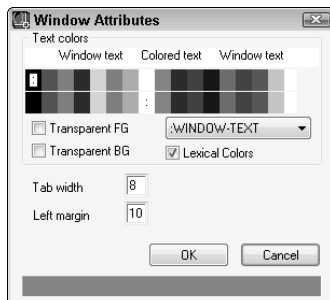
4. Click a color swatch in the bottom row to adjust the color to use for the background of the selected element.

The bottom row of color swatches controls the background color, which in most cases is the color white, except for highlighting, debugging, and comments.



5. To use color-coding, make sure that the **Lexical Colors** check box (in the Text Colors section) is selected .
6. Click **OK** to save the changes made in the Window Attributes dialog box.
7. Click **Yes** to save the color scheme as the default editor colors; click **No** to use the color scheme only while the text window is open.

**Figure BC4-3:** Adjust the colors used by the text window with the Window Attributes dialog box.





You can control the number of spaces that appear in the text window when the Tab key is pressed by adjusting the value in the Tab Width text box in the Window Attributes dialog box.



You can control the left margin of the text window with the value entered in the Left Margin text box in the Window Attributes dialog box. The value controls the amount of white space from the left edge of the text window to where the text first appears. The value is expressed in pixels.

### *Controlling text size and font style for the text window*

Changing the text size and font style for the text window adds a level of personalization to the Visual LISP Editor. Based on the screen resolution, you may need to change the size of the text for legibility. Follow these steps to change the size and font style used in the text window:

**1. In the Visual LISP Editor, click Tools⇨Window Attributes⇨Font.**

The Font dialog box (see Figure BC4-4) is displayed.

**2. In the Font list box, select a font.**

The Sample section updates with a preview of the current font settings.

**3. In the Font Style list box, select a font style.**

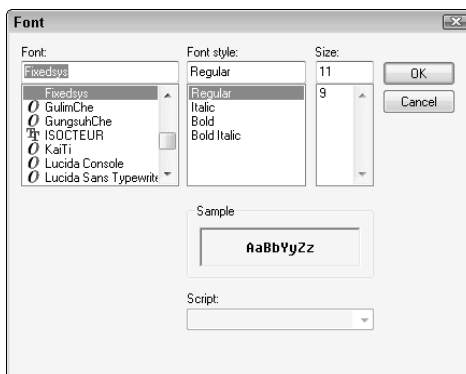
The Sample section updates with a preview of the current font style setting.

**4. In the Size list box, select a size.**

The Sample section updates with a preview of the current font size setting.

**5. Click OK to save the changes made in the Font dialog box and update the text window.**

**Figure BC4-4:**  
Adjust the text size and font style for the text window with the Font dialog box.



## Navigating the text window

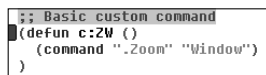
Navigating the text window doesn't require much of an explanation: It uses a lot of common Windows functionality that can be found in basic text editors such as Notepad. The text window can be maximized and minimized and contains both horizontal and vertical scroll bars, but it also contains some other nice options that you will not find in Notepad.

The Visual LISP Editor offers a few ways to improve navigation of both small and large source files. These navigation tools are in the Search menu. One option is called Bookmarks and another is called Go to Line:

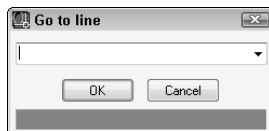
- ◆ **Bookmarks:** Bookmarks allow quick navigation between different sections of a source file. These markers remain active only while the file is open in the Visual LISP Editor. To create a bookmark, Click Search⇨Bookmarks⇨Toggle Bookmark. A dark green oval (see Figure BC4-5) appears in the left margin of the text window.  
To navigate between different bookmarks, select Next Bookmark or Previous Bookmark from Search⇨Bookmarks. Click Search⇨Bookmarks⇨Clear All Bookmarks to remove all the active bookmarks in the text window.
- ◆ **Go to Line:** Go to Line isn't as useful as it was in releases of AutoCAD prior to 2006. This is because in earlier releases, menu files needed to be edited outside AutoCAD by using a text editor. The Go to Line option was perfect for quickly jumping to a line in a menu file when an error was encountered during the compiling process.

The Go to Line option is still helpful when you need to find a problem in a DXF file when an error is encountered during the opening of the file. To use the Go to Line option, click Search⇨Go to Line. The Go to Line dialog box (see Figure BC4-6) is displayed. In the text field, enter the line number in the file you want to jump to and click OK.

**Figure BC4-5:**  
Bookmark in the left margin of the text window.



**Figure BC4-6:**  
Go here, go here.



## ***Creating a Basic Program***

Now that you have a taste for the Visual LISP Editor, I am sure you are just itching to get started. In the rest of this chapter, as you become familiar with the AutoLISP programming language, I will show you some of the other features of the Visual LISP Editor when it makes sense to introduce them.

### ***Creating a new AutoLISP file***

To start creating a basic program, make sure that a blank text window is available. The following steps explain how to create a blank text window and save the contents of the window to a new application file:

**1. In the Visual LISP Editor, click *Files*⇒*New File*.**

A new text window is created and is titled *<Untitled-0>*. The name of the file appears in the caption bar of the Visual LISP Editor if the text window is maximized or in the caption bar of the text window. The name of each new text window is incremented by a value of 1, so the next new text window is titled *<Untitled-1>*, and so on.

**2. In the Visual LISP Editor, click *Files*⇒*Save As*.**

The Save As dialog box is displayed, allowing you to specify a file name for the contents of the *<Untitled-0>* text window. By default, Visual LISP offers the name *tmp.lsp* for new application files.

**3. In the Save As Type drop-down list, specify the type of Lisp source file, if it is not already selected.**

The default file type is Lisp Source File.

**4. In the Save In drop-down list, specify the location where the application file should be saved.**

The Save In drop-down list allows you to select a folder or a named location such as My Documents to save the new file to.

**a. If the desired save to location is in a folder below the location that you selected in the Save In drop-down list, double-click the folder in the list box below the Save In drop-down list.**

The list box displays any folders or files located under the folder selected in the Save In drop-down list. The types of files that are displayed in the list box depend on the file filter selected in the Save As Type drop-down list.

**b. Keep double-clicking folders until you navigate to the folder that you want to save the new file to and type a new name in the File Name text box for the file.**

You should make the name as descriptive as you can without making it too long, just in case you need to find the file at a later date through Windows Explorer.

**5. Click Save.**

Clicking Save writes the contents of the text window to the file name and location that you specified. Note that the text window now displays the updated file name.

### Anatomy of an AutoLISP expression

To use the AutoLISP programming language, you must understand how it is structured and what makes up an AutoLISP expression. An AutoLISP expression must begin with either a left (open) parenthesis or an exclamation point:

- ◆ **(**: An open parenthesis is used to denote the start of an AutoLISP expression and must be balanced with a closing parenthesis to denote the end of the expression.
- ◆ **!**: An exclamation point is used to instruct the AutoLISP interpreter to return the value for the symbol that follows the exclamation point.

If you enter either an opening parenthesis or exclamation point at the command prompt in AutoCAD, it informs AutoCAD that the following information should be sent to the AutoLISP interpreter. The AutoLISP interpreter then evaluates and analyzes the expression(s) accordingly. If a return value is generated, it is displayed at the command line.

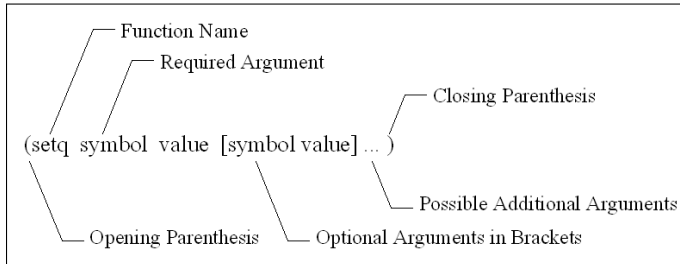
Table BC4-2 shows some sample AutoLISP expressions that you can type at the command prompt in AutoCAD and the expected results.

<b>AutoLISP Expression</b>	<b>Expected Result</b>
!XY	Returns the value of the XY variable, which by default is nil.
(command “_circle” “5,5” “10”)	Draws a circle at the coordinate 5,5 on the current layer with a radius of 10 units.
(setq X 4 Y 5 XY 9)	Sets the X variable to a value of 4; sets Y to a value of 5; and sets XY to a value of 9. The value 9 is returned to the command line because XY is the last variable that is set, and the setq function returns the value that it sets to a variable.
(+ 15 (* 2 12))	Returns a value of 39. The innermost functions are evaluated first in AutoLISP. The 2 multiplied by 12 is evaluated first for a value of 24. The value of 24 is then added to 15 to get an end result of 39.
(setq str (strcat “Welcome to \” “AutoLISP”))	The return value is “Welcome to AutoLISP”. The strcat function is used to concatenate string variables together.

## BC50 Creating a Basic Program

Now that you know how to identify an AutoLISP expression, the next step is to understand how an AutoLISP expression is structured when it starts with an open parenthesis. At a minimum, an AutoLISP expression must have an open (left) parenthesis, a function name, and a closing (right) parenthesis but can contain arguments as well (see Figure BC4-7). Usually AutoLISP expressions include arguments like many of the samples in Table BC4-2.

**Figure BC4-7:**  
The anatomy of an AutoLISP expression.



AutoLISP function and variable names are not case sensitive like function and variable names in other programming languages such as C++ or even VB. In other languages, a variable with the name A might represent something different than a variable with the name a, but not in AutoLISP.

### *Adding comments*

A comment is a string of information that describes a line or series of lines in an application file and does not get evaluated. A comment can identify what is taking place in a function or command, or can be a note at the top of the file that includes general revision or author information. A semicolon (;) is placed at the front of a text string to designate it as a comment. By default, a comment appears in purple text with a gray background in the Visual LISP Editor. A sample comment is

```
; New Custom command to create a dimension style
```

Along with using a semicolon to denote a comment for a single line of text, you can also use a semicolon and a pip symbol to create a paragraph comment. A sample of a paragraph comment is

```
; |
  Created on: 2/23/08
  Created by: Lee Ambrosius

  The following custom command can be used to
  configure the settings used for new drawings.
|;
```

## To command or just to function

When you are creating custom programs in AutoLISP, you have to make a decision whether the custom program will be used to define a new command or function. Typically, a command is used by entering its name at the AutoCAD command prompt or executing it from one of the user interface elements. Functions are not usually entered at the command prompt or with a user interface element but are used in an AutoLISP application file.

AutoLISP commands, on the surface, seem just like normal commands to users unless they try to use the commands in another AutoLISP program. AutoLISP commands are different than core commands defined with ObjectARX because they can't be used with the AutoLISP Command function. To create a command or function in AutoLISP, you use the function Defun, which stands for Define Function. The syntax for the Defun function follows:

```
(defun function_name ([arguments . . .] [/ variables . . .])
  expressions
  . . .
)
```

Here's a breakdown of the syntax:

- ◆ **Function\_filename:** The function or command name that can be typed at the command prompt or used in a command macro for a toolbar button. If the function name starts with c:, it indicates that a command is defined instead of a function.
- ◆ **Arguments:** Arguments are used to receive values from the calling function. Arguments are not used when defining a command.
- ◆ **Variables:** The variables that are defined with the Defun function are declared locally to the function or command. This is a way to manage variables that should exist only while the function or command is running. After the function or command ends, the variables are reset to the previous values they held.
- ◆ **Expressions:** An expression that uses the Defun function allows you to group AutoLISP expressions together with a specified name that can be used to reference those lines. Expressions represent the AutoLISP expressions that will be executed when the function or command runs.

An example of a custom function with the Defun function is

```
(defun RTD (rad)
  (* (/ rad PI) 180)
)
```

## BC52 *Creating a Basic Program*

---

The preceding example shows a custom function named **RTD** with a single argument named **rad**. (**RTD** is short for Radians to Degrees.) If you take a look at the expressions, you can see that there is a single expression using nested expressions. To convert a value in radians to degrees, the value of **rad** is divided by the value of **PI** and then multiplied by 180.

If you type **RTD** at the command prompt, you get a warning about an unknown command. This is because **RTD** is defined as a function, so it must be typed in at the command prompt with an opening and closing parenthesis and a value in radians like this: **(RTD 1.57079633)**. After the **RTD** function is evaluated, a value of 90 is returned to the command line.



There are three predefined variables in AutoLISP: **PI**, **T**, and **PAUSE**. **PI** holds the value 3.14159, **T** holds a non-nil value and evaluates to **T**, and **PAUSE** is used with the **Command** function to wait for the user to supply some input through the mouse or keyboard.

An example of creating a custom command with the **Defun** function is

```
(defun c:HL2 ()  
  (command ".line" "0,0" "2,0" "" )  
)
```

The preceding example shows a custom command named **HL2** with no arguments. **HL2** is short for Horizontal Line of 2 units. If you take a look at the expressions, you can see that there is a single expression with no nesting, but it is using multiple arguments. The function **Command** is used, which allows you to automate tasks by using the built-in AutoCAD commands.

In this case, the command that is being used is **LINE**. A line is being drawn from the coordinates 0,0 to 2,0. Because only a single line with starting and ending points is required, "" (two double quotation marks with no text between them) is used as the last argument to end the **LINE** command. The two double quotation marks is like pressing **Enter** on the keyboard. To run the custom command, all you need to do after it is loaded is to type **HL2** at the command prompt. You do not need to add an opening and closing parenthesis as you do when using a function.

### ***Creating your first AutoLISP program***

It's time to put everything together and create your first custom command. In this example, you create a new AutoLISP file, add a new command, and then load it into AutoCAD:

1. **On the menu browser or menu bar in AutoCAD, click Tools menu⇨AutoLISP⇨Visual LISP Editor.**

AutoCAD launches the Visual LISP Editor.

**2. In the Visual LISP Editor, click Files⇨New File.**

An empty text window titled <Untitled-0> is created.

**3. In the Visual LISP Editor, click Files⇨Save As.**

The Save As dialog box is displayed, allowing you to save the contents of the current text window.

**4. Make sure that the Save As Type drop-down list is set to Lisp Source File. In the File Name text box, enter MyStuff.LSP. Then in the Save In drop-down list, specify the save to location by selecting My Documents.**

**5. Click Save.**

The contents of the text window are written to a file named MyStuff.LSP located in the My Documents folder.

**6. In the text window titled MyStuff.LSP, enter the following code:**

```
;; Defines a two letter command to start a Zoom Window
(defun c:ZW ( )
  (command ".Zoom" "Window")
)
```

**7. In the Visual LISP Editor, click Files⇨Save.**

The new command is saved to the MyStuff.LSP file.

**8. Switch back to AutoCAD by clicking the icon in the Windows taskbar area or by clicking Activate AutoCAD from the Windows menu in the Visual LISP Editor.**

**9. At the command prompt, type ZW and press Enter.**

An error message is displayed in the command line window because the AutoLISP file hasn't been loaded yet:

```
Unknown command "ZW". Press F1 for help.
```

**10. Switch back to the Visual LISP Editor by clicking its icon in the Windows taskbar area. Or from the menu browser or menu bar in AutoCAD, click Tools menu⇨AutoLISP⇨Visual LISP Editor.**

**11. In the Visual LISP Editor, click Tools⇨Load Text in Editor.**

The custom command should now be ready to use in AutoCAD.

**12. Switch back to AutoCAD. At the command prompt, type ZW and press Enter.**

This time, no error message is displayed and the ZOOM command starts with the Window option automatically. The first prompt that is displayed is *Specify first corner*.

13. Select two points in the drawing window to complete the Window option of the ZOOM command.

### More Than Just the Essentials of AutoLISP

Now that you have a good foundation of what it takes to create an AutoLISP file and a custom command, it's time to introduce you to some more functions. AutoLISP can perform math calculations, string manipulation, and much more. All of these are important in adding functionality to your custom programs.

#### Supported data types

AutoLISP is a programming language, so it uses data types to represent certain types of information. Data types play a role in how your application exchanges information with functions and even with AutoCAD and the user. Table BC4-3 shows the different data types in AutoLISP.

Table BC4-3		Data Types in AutoLISP
Data Type	Description	Example
Integer	A whole number with no decimal place, and values fall in the range of -32,768 and 32,767	10
Real	Any number with a decimal place and 14 significant places of precision	1.0, 0.23451, .5
String	Any number of alphanumeric characters enclosed in double quotes	"Hello"
List	An expression enclosed in parentheses	(0.5 0.5 0), ("Hello" "World"), (1 "Hello")
Symbol	Variables used to hold data	PI, abc, retVal
Selection	A valid selection set to hold references to entities	<Selection set: 1>set
Entity name	A valid entity name used to reference an individual entity in the drawing	<Entity name: 0cf0121>
VLA-object	A valid COM object used with ActiveX automation	#<VLA-OBJECT IAcadApplication 00c2db8c>
File descriptor	A reference point to an open file in memory	#<file "acad.pgp">

## ***Math functions***

Because AutoCAD is a drafting program, you most likely will find yourself needing to work with numbers from time to time. AutoLISP math functions come in a variety of forms, from standard math functions such as + (addition) and - (subtraction) to much more complex functions, such as sin (sine) and atan (arctangent).

Table BC4-4 shows some of the different math functions in AutoLISP.

<b>Function Syntax</b>	<b>Description and Example</b>
(+ [ <i>number number ...</i> ])	Sums all numbers and returns the value Example: (+ 1 2.5) Result: 3.5
(- [ <i>number number ...</i> ])	Subtracts all the numbers from the first one and returns the value Example: (- 1 0.25) Result: 0.75
(* [ <i>number number ...</i> ])	Multiplies all the numbers together and returns the value Example: (* 1 3 5) Result: 15
(/ [ <i>number number ...</i> ])	Divides the first number by the product of all the remaining numbers and returns the value Example: (/ 2 5) Result: 0
(abs <i>number</i> )	Returns the absolute value Example: (abs -1.34) Result: 1.34
(sqrt <i>number</i> )	Returns the square root of the number Example: (sqrt 3) Result: 1.73205

## ***String functions***

Strings provide feedback to the user in the form of error messages or give feedback on the current status of a command if it takes a while to run. Another thing that strings are useful for is command prompts. Because AutoCAD is still primarily command prompt based, you will most likely be asking the user of your custom programs to provide information at the command prompt.

Table BC4-5 shows some of the different string manipulation functions that are available in AutoLISP.

Table BC4-5	Some of the Basic String Manipulation Functions
<i>Function Syntax</i>	<i>Description and Example</i>
(strcase <i>string</i> [flag])	Returns the string in either uppercase or lowercase. If the optional flag value is set to T, the string is returned in lowercase. Example: (strcase "Hello") Result: "HELLO"
(strcat [ <i>string</i> . . .])	Concatenates all the supplied strings together and returns a single string. Example: (strcat "Hello" " " "World") Result: "Hello World"
(substr <i>string start</i> [ <i>length</i> ])	Returns a portion of the string based on the start position and the length supplied. Example: (substr "Hello World" 7) Result: "World"
(vl-string-subst <i>replace_string pattern string</i> [start])	Looks for the pattern in the string and replaces it with the <i>replace_string</i> value, and returns the result of the replace. Example: (vl-string-subst "Planet" "World" "Hello World") Result: "Hello Planet"

### List functions

Lists are everywhere in AutoLISP: You've been working with lists and weren't even aware of it. Lists are used to represent groups of things, such as an expression that contains a function and an argument. Lists are used also to represent coordinate values and even entity properties.

Table BC4-6 shows some of the different list functions in AutoLISP.

Table BC4-6	Some Basic List Functions
<i>Function Syntax</i>	<i>Description and Example</i>
(list [ <i>expression</i> . . .])	Returns a list of all the provided expressions. Example: (list 1.3 12.75 0) Result: (1.3 12.75 0)
(car <i>list</i> )	Returns the first item in a list. There are combinations of car and cdr to get items in other places of the list, such as cadr. Example: (car (list 1.3 12.75 0)) Result: 1.3
(cdr <i>list</i> )	Returns all items in a list except the first one. There are combinations of car and cdr to get items in other places of the list, such as cadr. Example: (cdr (list 1.3 12.75 0)) Result: (12.75 0)

<i>Function Syntax</i>	<i>Description and Example</i>
(nth <i>index list</i> )	Returns the item in the location of the list specified by the value of <i>index</i> . The first item in the list is index 0. Example: (nth 2 (list 1.3 12.75 0)) Result: 0
(reverse <i>list</i> )	Reverses the item order of the list and returns the new list. Example: (reverse (list 1.3 12.75 0)) Result: (0 12.75 1.3)

### ***Data conversion functions***

As you start using more AutoLISP functions, you'll want to use the results of one AutoLISP function with another, but each AutoLISP function requires you to use specific data types for its arguments. In these situations, you need to convert values from one data type to another. AutoLISP contains some functions specifically for this purpose.

Table BC4-7 shows some of the different conversion functions in AutoLISP.

<b>Table BC4-7                      Some Basic Conversion Functions</b>	
<i>Function Syntax</i>	<i>Description and Example</i>
(atof <i>string</i> )	Returns the string as a real number. Example: (atof "12.5") Result: 12.5
(atoi <i>string</i> )	Returns the string as an integer. Example: (atoi "1.75") Result: 1.75
(itoa <i>int</i> )	Returns the integer as a string. Example: (itoa 12) Result: "12"
(rtos number [mode [precision]])	Returns the real number into a formatted string. Mode is a value of 1–5 and precision is 0–8, the same as LUPREC. Mode: 1: Scientific 2: Decimal 3: Engineering 4: Architectural 5: Fraction Example: (rtos 2.5 2 8) Result: "2.50000000"

### *Saving and accessing values for later*

Now that you have all these different values being returned by math, string, list, and conversion functions, it would be nice to save the values for use later in your custom programs. AutoLISP offers some different ways to store and retrieve values. Some of the ways affect how AutoCAD features behave or can be used to temporarily store information that can be recalled later in your program, such as a calculation or user input that might be needed for another function.

Table BC4-8 shows some of the different data storage and retrieval functions in AutoLISP.

<b>Table BC4-8</b>	<b>Data Storage and Retrieval Functions</b>
<i>Function Syntax</i>	<i>Description and Example</i>
(setq <i>variable value</i> [ <i>variable value</i> ] ...)	Sets the value to a variable. The variables only exist while the drawing is open. Example: (setq msg "Error occurred") Result: "Error occurred"
(setvar <i>variable_name value</i> )	Sets the value to a system variable. Example: (setvar "clayer" "0") Result: "0"
(getvar <i>variable_name</i> )	Returns the current value of a system variable. Example: (getvar "clayer") Result: "0"

### *Exchanging information with AutoCAD*

One of the hardest tasks in drafting is the initial setting up and managing of CAD standards. In the following steps, you create a custom command that draws a circle on a specific layer and sets it to a specified diameter:

- 1. On the menu browser or menu bar in AutoCAD, click Tools menu⇨AutoLISP⇨Visual LISP Editor.**

AutoCAD launches the Visual LISP Editor.

- 2. In the Visual LISP Editor, click Files⇨Open File.**

The Open File to Edit/View dialog box is displayed, allowing you to open an existing file.

- 3. Browse to the My Documents folder and select the MyStuff.LSP file.**

If the MyStuff.LSP file is not in the My Documents folder, make sure that you complete the procedure in the "Creating your first AutoLISP program" section earlier in this chapter before continuing.

**4. Click Open.**

The contents of the MyStuff.LSP file is loaded into a text window.

**5. In the text window, place the cursor behind the last parenthesis and press Enter twice so there is a blank line below the ZW command.**

**6. In the text window, enter the following expressions and command:**

```
;; Custom command to create a new layer and draw a
;; 0.5 diameter circle.
(defun c:BCIRC ( / cur-layer)
  ;; Store the current layer
  (setq cur-layer (getvar "clayer"))
  ;; Create a new layer called Hole and
  ;; set it current
  (command ".layer" "m" "Hole" "c" "5" "" "")
  ;; Draw a 0.5 diameter circle and ask the user for
  ;; the center point
  (command ".circle" PAUSE "d" 0.5)
  ;; Restore the original layer
  (setvar "clayer" cur-layer)
  ;; Stop an return values from being echoed
  (princ)
)
```

**7. In the Visual LISP Editor, click Files→Save.**

The new command is saved to the MyStuff.LSP file.

**8. In the Visual LISP Editor, click Tools→Load Text in Editor.**

The custom command is loaded into AutoCAD and is ready to be used.

**9. Switch back to AutoCAD.**

**10. At the command line, type BCIRC and press Enter.**

You are prompted for the center point of the circle. A new circle object is generated on the Hole layer, which is blue. The circle has a diameter of 0.5. The program restores the previous layer before the CIRCLE command was started. If the user presses the Escape key, the previous layer will not be restored. To resolve this, modify the command so that it looks like the following code:

```
;; Custom command to create a new layer and draw a
;; 0.5 diameter circle.
(defun c:BCIRC ( / cur-layer)
  ;; Store the current layer
  (setq cur-layer (getvar "clayer"))
  ;; Create a new layer called Hole and
  ;; set it current
  (command ".layer" "m" "Hole" "c" "5" "" "s" cur-
layer "")
  ;; Draw a 0.5 diameter circle and ask the user for
  ;; the center point
```

## BC60 Getting Information to and from the User

---

```
(command ".circle" PAUSE "d" 0.5)
;; Change the layer of the new circle
(command ".change" "l" "" "p" "la" "Hole" "")
;; Stop an return values from being echoed
(princ)
)
```

With the revised code, the current layer is never changed; instead, the object's properties are modified with the CHANGE command. Although the second solution requires more code, it is better because the drawing properties are never changed. If the user presses the Escape key, the current layer will not be affected.

## Getting Information to and from the User

User input comes in a variety of ways in AutoCAD. Typical user input in AutoCAD is done through the command prompt and the mouse. AutoLISP is full of functions for collecting user input.

Table BC4-9 shows some of the user input functions in AutoLISP.

<b>Table BC4-9</b>	<b>Some Basic User Input Functions</b>
<i>Function Syntax</i>	<i>Description and Example</i>
(getpoint [ <i>point</i> ] [ <i>message</i> ])	Requests the user to enter a point or select one in a drawing Example: (getpoint "\nPick first point: ") Result: (21.9975 13.1977 0.0)
(getcorner point [ <i>message</i> ])	Requests the user to enter a point or select one in a drawing for the opposite corner of a rectangle Example: (getpoint '(0 0) "\nPick opposite corner: ") Result: (24.4237 14.9183 0.0)
(getdist [ <i>point</i> ] [ <i>message</i> ])	Requests the user to pick one or two points to calculate the overall distance between the two points picked Example: (getdist "\nPick two points: ") Result: 7.93347
(getint [ <i>message</i> ])	Requests the user to enter a whole number Example: (getint "\nEnter your age: ") Result: 29
(getreal [ <i>message</i> ])	Requests the user to enter any number with or without decimal places Example: (getreal "\nEnter a number: ") Result: 32.6

## Giving feedback to the user

Programming is all about give and take, so you should make sure that you are not just taking input from your users without also giving them feedback. Feedback can be in the form of text at the command prompt or even an alert message box when an error occurs.

Table BC4-10 shows some of the different user feedback functions in AutoLISP.

<b>Table BC4-10</b>	<b>Some Basic User Feedback Functions</b>
<i>Function Syntax</i>	<i>Description and Example</i>
<i>(alert message)</i>	Displays a message box with an OK button in it. Example: (alert "Hello World")
<i>(prompt message)</i>	Displays a text string in the command line. Example: (prompt "\nHello World") Result: Hello Worldnil
<i>(princ [expression [filedescriptor]])</i>	Prompt returns the value of nil, along with many other AutoLISP functions. You can keep AutoCAD from echoing nil and the last return value by using a function called princ. Optionally, princ can also be used to display a text string or write a text string to a file. When princ is used to display a text string, it may display the text string twice because it displays and echoes the message. It is best not to use princ to display a text string at the command line. Example: (prompt "\nHello World")(princ) Result: Hello World  Example: (princ "\nSelect object:") Result: Select object: "\nSelect object:"
<i>(textscr)</i>	Displays the AutoCAD text window. Example: (textscr)
<i>(graphscr)</i>	Displays the AutoCAD graphics screen. Example: (graphscr)

## Other functions to note

As you might have already noticed, a lot of functionality is packed into the AutoLISP programming language. The functions I cover in this section can help to make your custom programs much more robust.

Table BC4-11 shows some of the other functions available in AutoLISP.

## BC62 Getting Information to and from the User

---

Table BC4-11	Some Other Functions
<i>Function Syntax</i>	<i>Description and Example</i>
<code>(while testcondition[expression ...])</code>	<p>Evaluates the test condition. If it is true, the expressions are evaluated repeatedly until the test condition is false.</p> <p>Example: <code>(setq cnt 10)(while (&gt; cnt 0)(prompt(strcat "\nCount down: T-"(itoa cnt) ))(prompt "\nBlast off"))</code></p> <p>Result:</p> <p>Count down: T-10 Count down: T-9 Count down: T-8 Count down: T-7 Count down: T-6 Count down: T-5 Count down: T-4 Count down: T-3 Count down: T-2 Count down: T-1 Blast off</p>
<code>(if testcondition then expression [else expression])</code>	<p>Evaluates the test conditional expression and evaluates one of the two expressions. If the test conditional expression is true, the <i>then</i> expression is evaluated; if not, the <i>else</i> expression is evaluated.</p> <p>The function <code>progn</code> is used to group more than one AutoLISP expression together because <code>IF</code> can only handle one expression for the <i>then</i> and <i>else</i> expressions.</p> <p>Example:</p> <pre>(setq XY 3.1) (if (&gt; XY 3)   (progn     (prompt "\nNumber greater than 3.")     (prompt "\nanother line...")   )   (prompt "\nNumber less than 3.") ) </pre> <p>Result: Number greater than 3.</p>

<i>Function Syntax</i>	<i>Description and Example</i>
<code>(ssget [selection-method] [point1 [ point2]] [point-list] [filter-list])</code>	<p>Allows the user to select objects on-screen similar to commands like copy and move, but ssget has special built-in selection methods and filtering to give your custom commands and programs some additional control. There are also functions like sslength and sssize, which are specific to working with the returned selection set value.</p> <p>Example:</p> <pre>(setq SS (ssget)) (if (/= SS nil)   (prompt     (strcat "\nYou selected "       (itoa (sslength SS))       " objects." )   )   (prompt "\nNo objects selected.")) )</pre> <p>Result: You selected 7 objects.</p>
<code>(entsel [message])</code>	<p>Allows the user to select only an individual object. The return value is a list of the points selected on-screen and the entity name of the object selected.</p> <p>Example:</p> <pre>(setq ENT (entsel "\nSelect a Circle: ")) (if (/= ENT nil)   (progn     (if (= (cdr (assoc 0 (entget (car ENT)))) "CIRCLE")       (prompt "\nA Circle was selected.")       (prompt "\nA Circle was not selected."))     )   )   (prompt "\nNo object was selected.")) )</pre> <p>(princ)</p> <p>Result: A circle was not selected.</p>

## *Using the Debug Tools in the Visual LISP Editor*

One of the most challenging aspects of programming is the process of debugging a custom program. *Debugging* is the process of looking for errors in a program and correcting them. These problems in a program are referred to as *defects*, or *bugs*. A defect is simply something that happens in a program that was not intended. The debugging process can be short and simple in the beginning, but the process can become much longer and more difficult as your programs evolve. The Visual LISP Editor provides several tools that can help make the debugging process go smoothly.

### Breakpoints

Breakpoints identify locations in your custom programs where you would like to stop and evaluate what is happening. This allows you to locate and fix defects in a routine much faster than having to step through the entire program again and again. A breakpoint is represented by a parenthesis with a red background (see Figure BC4-8) and is available between AutoCAD sessions — if you close and reopen AutoCAD, the breakpoints are still available the next time you open the LISP file in the Visual LISP Editor.

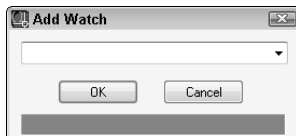
**Figure BC4-8:**  
Breakpoint in the text window.

```
:: Demonstrates how to create a new layer and to  
:: make sure that a 0.5 diameter circle is created  
:: on the new layer.  
(defun c:BCIRC ()  
  (setq cur-layer (getvar "clayer"))  
  (command ".layer" "m" "Hole" "c" "5" "" "" )  
  (command ".circle" PAUSE "d" 0.5)  
  (setvar "clayer" cur-layer)  
  (princ)  
)
```

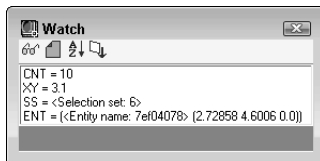
### Watch what is happening

*Watch* is a nice process to use when you are getting an unexpected result during the execution of a custom program. Watch is used to evaluate the current value of a variable or the return value of an expression so you can validate when an incorrect value is being returned. Using watch is a two-step process: First you set up a watch through the Add Watch dialog box (see Figure BC4-9), and then you monitor the changes in the Watch dialog box (see Figure BC4-10).

**Figure BC4-9:**  
Setting up a watch.



**Figure BC4-10:**  
Keeping a watch out for strange activity.



## Setting up breakpoints and using watch

In this procedure, you set a breakpoint and a watch to monitor the current value of a variable during the execution of the custom command BCIRC. If you didn't create the MyStuff.LSP file with the BCIRC command back under the section titled "Exchanging information with AutoCAD," you should go back to that section and perform the steps there first before continuing here:

- 1. Launch the Visual LISP Editor and open the MyStuff.LSP file from the My Documents folder.**

The MyStuff.LSP file is opened in a text window in the Visual LISP Editor.

- 2. Position the cursor over the cur-layer variable in the custom command BCIRC and double-click.**

The cur-layer variable is highlighted.

- 3. Right-click over the variable and select Add Watch.**

The cur-layer variable is added to the Watch dialog box and its current value is nil (see Figure BC4-11). As the custom command is being executed, you are able to see the value of the variable change as it is being set to a different value.

- 4. Position the cursor over the left parenthesis of the first line that starts with the Command function and right-click.**

The right-click menu for the text window is displayed.

- 5. From the right-click menu, select Toggle Breakpoint.**

The left parenthesis is highlighted with a red background, denoting the breakpoint has been set.

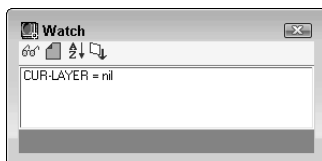
- 6. In the Visual LISP Editor, click Tools⇨Load Text in Editor.**

The custom command is loaded into AutoCAD and is ready to be used.

- 7. Switch to AutoCAD and type BCIRC at the command prompt and press Enter to run the custom command.**

Focus switches back to the Visual LISP Editor and highlights the line with the breakpoint (see Figure BC4-12). Notice that the cur-layer variable in the Watch dialog box is no longer assigned nil, because it currently holds the value of the CLAYER system variable.

**Figure BC4-11:**  
A variable added to the Watch dialog box.



**Figure  
BC4-12:**

Evaluation of the custom command is paused at the line with the breakpoint.

```
;; Custom command to create a new layer and draw a  
;; circle with a diameter of 0.5  
(defun c:BCIRC ( / cur-layer)  
  
  ;; Stores the current layer  
  (setq cur-layer (getvar "clayer"))  
  
  ;; Creates a new layer called Hole and sets it current  
  (command ".layer" "n" "Hole" "c" "5" "" "")  
  
  ;; Draws a circle with a diameter of 0.5
```

**8. Press F8 on the keyboard to resume stepping through the next expression after the breakpoint.**

The next line or expression is highlighted, allowing you to evaluate what is happening line by line in the watch window and in AutoCAD.

**9. Press F8 a few more times.**

The next line or expression is highlighted and evaluated.

**10. Press CTRL+F8 to resume normal execution of the custom program.**

The routine finishes executing.



One thing that can be helpful when you are getting started is to add expressions in your custom program that use the Prompt or Alert functions so you know which expression is being executed. This can be much less distracting than stepping through expressions by using the Visual LISP Editor when testing your custom programs.

### ***AutoLISP error messages***

As you start to create your own custom programs with AutoLISP, you'll certainly encounter some problems. AutoCAD lets you know that there's a problem with your custom programs, but doesn't clearly explain how to identify or fix the problem. This can be frustrating, but remember that you have breakpoints and the Watch dialog box at your disposal to help you identify what is going wrong.

Table BC4-12 shows some of the more common errors that AutoCAD displays when it can't evaluate an AutoLISP expression.

<b>Table BC4-12</b>	<b>Some Common AutoLISP Error Messages</b>
<i>Error Message</i>	<i>Description and Example</i>
AutoCAD rejected function	One of the functions that you are using in your custom program can't be used in the current context. This could be related to functions such as Setvar, with a read-only variable, Tblnext using an invalid table entry, or if one of the Getxxx functions for getting input is used in the Command function. Example: (setvar "ACAD" "c:\\") ; error: AutoCAD variable setting rejected: "ACAD" "c:\\"
Bad argument \type	An incorrect data type was passed into an argument. Example: (setq strName nil)(strcat "Hello" strName) ; error: bad argument type: stringp nil
(_>	Extra opening parenthesis or an unbalanced one. This can be one of the hardest errors to find if your program is large. Example: ((strcat "Hello" " World!") (_>
extra right paren	Extra closing parenthesis or an unbalanced one. Like the extra opening parenthesis, this can be hard to find if your program is large. Example: (strcat "Hello" " World!") ; error: extra right paren on input
("_>	A double quote mark that represents a string is missing. Example: (strcat "Hello" " World!") ("_>
too few arguments	Required arguments for a function are missing. Example: (alert) ; error: too few arguments
too many arguments	There are too many arguments for the function that is being used. Example: (prompt "Hello" " World!") ; error: too many arguments



If you're having problems with unbalanced parentheses in your custom programs, the Visual LISP Editor offers a Parentheses Matching tool in the Edit menu. This can help you identify where the problem with parentheses exists.

## *Going GUI with DCL*

AutoLISP was designed to be used as a way to automate repetitive tasks and to create custom commands to complete a task that could not normally be completed easily by using AutoCAD. As technology moved forward, AutoLISP needed to evolve from a command line utility to having the ability to utilize a dialog box for managing settings and collecting information from a user. To do this, AutoLISP utilizes a language called DCL — Dialog Control Language. This language is used to lay out controls for a dialog box.

### *Basics of DCL*

DCL files are plain text files but must follow a specific structure and definition system. Two definition systems, dialog boxes and tiles, make up the controls that should be displayed. Each of the tiles and the dialog box have different attributes that can be set to control the appearance and functionality of the tiles and dialog boxes. An example of a dialog box structure looks like this:

```
MYDLG : dialog { label = "My Dialog";  
               <Tiles go here>  
}
```

The structure that a tile (or control) follows is similar to that of a dialog box. An example of the structure for a Toggle tile is

```
: toggle { label = "Use second edit box, too.";  
           value = "0";  
}
```

Table BC4-13 shows some of the common tiles used in DCL.

<b>DCL Tile</b>	<b>Description and Example</b>
Dialog	Defines a dialog box object and is used as a container to hold different tiles Example: MYDLG : dialog { label = "My Dialog";}
Button	Creates a push button or command button with a caption Example: : button { label = "Import..."; action = "(btn_import)"; key = "btn_import"; mnemonic = "I"; }

<i>DCL Tile</i>	<i>Description and Example</i>
List_box	Creates a vertical list of strings that can be selected individually or as multiples Example: : list_box { action = "(list_clicked)"; list = "Yes\nNo\nUnknown"; key = "list"; value = "Unknown"; }
Ok_only	Creates an OK button only Example: ok_only;
Ok_cancel	Creates an OK and Cancel button combination Example: ok_cancel;
Popup_list	Creates a drop-down list Example: : popup_list { action = "(pop_clicked)" list = "Yes\nNo\nUnknown"; key = "pop"; value = "Unknown"; }
Radio_button	Creates a radio or option button Example: : radio_button { label = "On"; action = "(radio_clicked)"; key = "radio_on"; }
Toggle	Creates a check box Example: : toggle { label = "Use second edit box, too."; action = "(toggle_ed2)"; value = "0"; }

### *Adding comments*

A comment is a line or series of lines in the DCL structure that do not get interpreted. A comment can be used to identify what is taking place at a location in a dialog box or can be a general note at the top of the file that includes general revision or author information. Double forward slashes are placed in front of a text string to designate it as a comment. By default, a

comment appears in purple text with a gray background in the Visual LISP Editor. A sample comment is

```
// Created on February 23, 2008 by Lee Ambrosius
```

### *Using DCL to add interaction to AutoLISP*

DCL needs AutoLISP to do anything: A DCL file itself doesn't do anything other than take up space on a disk as a file. In these steps, I show you how to display a dialog box with AutoLISP and add an action to a tile to make it interactive:

- 1. Launch the Visual LISP Editor and open the file MyStuff.LSP from the My Documents folder.**
- 2. Create a new file with the name MyStuff.DCL and make sure that it's saved as the type DCL Source File instead of LSP Source File.**

A new text window is created titled MyStuff.DCL.

- 3. In the MyStuff.DCL text window, enter the following dialog box and tile definitions:**

```
// Dialog box is a support file of MyStuff.lsp
MYSTUFF :dialog { label = "MyStuff Example";
  : toggle { label = "Use second edit box, too?";
    value = "0";
    key = "tg11";
  }
  :boxed_column { label = "Edit Box group";
    : edit_box { label = "1st Edit Box";
      key = "edBox1";
    }
    : edit_box { label = "2st Edit Box";
      key = "edBox2";
      is_enabled = "False";
    }
  }
  ok_cancel;
}
```

- 4. In the Visual LISP Editor, click Files↔Save.**

The new dialog box and tile definitions are saved to the MyStuff.DCL file.

- 5. In the Visual LISP Editor, click Tools↔Interface Tools↔Preview DCL in Editor.**

The Enter the Dialog Name dialog box (see Figure BC4-13) is displayed. This dialog box allows you to control which dialog box defined in the DCL file should be previewed.

**Figure BC4-13:**  
Which dialog box should be previewed?



**6. In the down-down list, select MYSTUFF and click OK.**

The MyStuff Example dialog box (see Figure BC4-14) is displayed, but if you click on any of the tiles, nothing happens because no AutoLISP code is defined to interact with the tiles in the dialog box.

**Figure BC4-14:**  
Preview of the MyStuff Example dialog box.



**7. Click any control to return to the Visual LISP Editor.**

The preview of the MyStuff Example dialog box closes and you return to the Visual LISP Editor.

**8. In the Visual LISP Editor, click Window → MyStuff.LSP or reopen the file if it isn't open.**

A text window with MyStuff.LSP is displayed.

**9. In the MyStuff.lsp text window, enter the following custom command:**

```
;; Custom command to display the MyStuff Example
;; dialog box.
(defun c:MS_DCL ()
  ;; Function executed when the OK
  ;; button is clicked
  (defun do_exit ()
    (done_dialog 1)
  )
)
```

```
;; Function executed when the Cancel button is
clicked
(defun do_cancel ()
  (done_dialog 0)
)
;; Function executed when the Toggle is clicked
(defun edit_tgl ()
  ;; Get the current value of the Toggle
  (setq EDTGL (get_tile "tgl1"))
  ;; If the Toggle is checked then enabled the
  ;; second text box.
  (if (= EDTGL "1")
    (mode_tile "edBox2" 0)
    (mode_tile "edBox2" 1)
  )
)
;; Load the MyStuff Example dialog box into memory
(setq dld (load_dialog "MYSTUFF.DCL"))
;; Check to see if the dialog box was loaded
(if (> dld 0)
  (prong
    ;; Create a reference to the dialog box
    (new_dialog "MYSTUFF" dld)
    ;; Set the actions for the OK and
    ;; Cancel buttons
    (action_tile "accept" "(do_exit)")
    (action_tile "cancel" "(do_cancel)")
    ;; Enable the dialog box
    (start_dialog)
    ;; Unload the dialog box
    (unload_dialog dld)
  )
)
)
```

### 10. In the Visual LISP Editor, click **Tools** → **Load Text in Editor**.

The new custom command MS\_DCL is loaded into AutoCAD and ready to use.

### 11. Switch to AutoCAD. Type MS\_DCL at the command prompt and press **Enter**.

One of two things will happen: The dialog box will or will not be displayed. If the dialog box is not displayed, check the expressions that are part of the MS\_DCL command. If they seem fine, make sure that the MyStuff.DCL file is in one of the AutoCAD Support File Search Paths.



If you are sure that the file is in one of the AutoCAD Support File Search Paths, you can use the AutoLISP FindFile function to find the file. Following is an example of this function:

```
(findfile "acad.pgp")
"C:\\Documents and Settings\\<user name>\\Application
  Data\\Autodesk\\AutoCAD
  2009\\R17.2\\enu\\support\\acad.pgp"
```

Usually by default, AutoCAD does find files in My Documents; however, that can vary from install to install. You can either add the My Documents folder to the Support File Search Path under the Files tab of the Options dialog box or move the file to one of the folders listed there.

**12. Click the toggle (check box) and note that nothing happens.**

There is a function defined as edit\_tgl, but it is not assigned to the toggle.

**13. Below the AutoLISP expression (action\_tile “cancel” “(do\_cancel)”), add this expression:**

```
(action_tile "cancel" "(do_cancel)")
(action_tile "tgl1" "(edit_tgl)")
(start_dialog)
```

The key from the DCL file is how you associate a function with a tile. You can also use the action attribute for the tile in the DCL file.

**14. Reload the AutoLISP file by using the Load Text in Editor option and type MS\_DCL at the command prompt of AutoCAD. Click the toggle again to see what happens.**

This time, the toggle should enable the second edit box through the edit\_tgl function that was defined in the MS\_DCL command.

## *Using ActiveX Automation with AutoLISP*

Using ActiveX automation with AutoLISP allows you to do things more efficiently and access things that you just can't normally do with AutoCAD commands and AutoLISP functions alone. One thing that you normally can't do with AutoLISP is to manipulate the AutoCAD application or other open documents. With ActiveX automation, you can even create objects and change their properties without the need to issue any commands, like CHANGE. To enable the ActiveX automation features of AutoLISP, use the function VL-Load-Com. The VL-Load-Com function doesn't take any arguments.

### *Referencing the AutoCAD application*

The AutoCAD application object is the gatekeeper for accessing the objects that represent all open drawings. The AutoCAD application object is retrieved by the function Vlac-Get-Acad-Object. This function does not require any arguments, but it does return a reference to an AcadApplication object, which is the object that represents the AutoCAD application. With this object, you can access other properties specific to the application object such as its visibility and position on-screen.

### Where to go from here

If you are wondering whether this is all there is to AutoLISP, the short and simple answer is no. You can find a lot of information in the AutoCAD Online Help system under the topics AutoLISP,

Visual LISP, and DXF. Another great resource is the Internet. You can even find Web sites that are dedicated to tutorials on AutoLISP that you can work through.



When you have a reference to an object, you can use the function `Vlax-Dump-Object`. This function requires one argument: a reference to an object like `AcadApplication`. The function also has an optional argument that allows you to see just the properties of the object or both the properties and methods. An example of how the `Vlax-Dump-Object` function is used follows:

```
(vlax-dump-object (vlax-get-acad-object) T)
```

If the example is entered at the command prompt in AutoCAD, it displays all the properties and methods associated with the AutoCAD application in the AutoCAD text window.

### Using methods of an object

Methods in ActiveX automation are, for the most part, the same as an AutoLISP function. They can return a value or not based on how the method is set up. Methods are called in a different way in ActiveX than for an AutoLISP function because AutoLISP is not able to communicate directly with ActiveX automation.

To use a method, you use the function `Vlax-Invoke-Method`. This function requires at least two arguments. The first argument is the object that you are calling the method on, and the second is the method that you want to call on an object. The method might require additional parameters.

An example of calling a method for an object is

```
(vlax-invoke-method (vlax-get-acad-object) 'ZoomAll)
```

In this example, the `ZoomAll` method is being called on the `AcadApplication` object that is returned from the `Vlax-Get-Acad-Object` method.

### Setting and retrieving a property of an object

Properties in ActiveX automation are attributes of an object that define the characteristics of an object — these properties can be updated or retrieved.

The processes closest in AutoLISP to working with properties are the functions Entget/Entupd and Setvar/Getvar, which allow you to interact with an object and settings. After you understand the process of modifying properties on an object, you can do things that you just can't normally do with AutoLISP alone.

To retrieve the value of a property, you use the function Vlx-Get-Property; to set the value of a property, you use the function Vlx-Put-Property. The Get function requires two arguments, and the Put function requires three arguments. The first argument for both Get and Put is the object that you are interested in working with, and the second is the property name. In the case of the Put function, the third argument is the value you want to assign to the property.

An example of retrieving a property from an object is

```
(vlax-get-property (vlax-get-acad-object) 'FullName)
```

An example of setting a property for an object is

```
(vlax-put-property (vlax-get-acad-object) 'WindowState 2)
```

### ***Revising the BCIRC command***

Using ActiveX automation with AutoLISP adds a new level of complexity to your program, but the ability to use ActiveX automation with AutoLISP is great when you realize that you can enhance your existing custom programs without the need to rewrite them in a different language like VBA. Earlier in the chapter, you created the custom command BCIRC, which added a new layer to the drawing and asked the user to pick a point at which to draw a 0.5 diameter circle. The following steps do the same thing with ActiveX automation:

- 1. Launch the Visual LISP Editor and open the file MyStuff.LSP from the My Documents folder.**
- 2. In the text window, place the cursor behind the last parenthesis on the very last function/command in the file and press Enter twice so there is a blank line at the beginning.**
- 3. In the text window, enter the following expressions and command:**

```
;; Initiate ActiveX Automation  
(vl-load-com)  
;; Custom command to create a new layer and draw a  
;; 0.5 diameter circle.
```

## BC76 Using ActiveX Automation with AutoLISP

---

```
(defun c:BCIRC-AX ( / acadObj docObj mspaceObj curlayer
  layersObj newLayer colorObj circObj)
  ;; Get a reference to the AutoCAD object
  (setq acadObj (vlax-get-acad-object))
  ;; Get a reference to the active drawing
  (setq docObj (vlax-get-property acadObj
  'ActiveDocument))
  ;; Get a reference to Model Space
  (setq mspaceObj (vlax-get-property docObj
  'ModelSpace))
  ;; Get a reference to the active layer
  (setq curlayer (vlax-get-property docObj
  'ActiveLayer))
  ;; Get a reference to the Layers table of the
  ;; current drawing
  (setq layersObj (vlax-get-property docObj 'Layers))

  ;; Create the new layer named Hole
  (setq newLayer (vlax-invoke-method layersObj 'Add
  "Hole"))
  ;; Assign the ACI Color Blue to the new layer
  (setq colorObj (vlax-get-property curlayer
  'TrueColor))
  (vlax-put-property colorObj 'ColorMethod
  acColorMethodByACI)
  (vlax-put-property colorObj 'ColorIndex acBlue)
  (vlax-put-property newLayer 'TrueColor
  colorObj)
  ;; Get the center point for the circle
  (setq utilityObj (vlax-get-property docObj
  'Utility))
  (setq centerPoint (vlax-invoke-method utilityObj
  'GetPoint nil "\nSelect the Center Point: "))

  ;; Create the new circle in the drawing
  (setq circObj (vlax-invoke-method mspaceObj
  'AddCircle centerPoint 0.25))
  (vlax-put-property circObj 'Layer "Hole")
  ;; Stop an return values from being echoed
  (princ)
)
```

#### **4. In the Visual LISP Editor, click **Files**→**Save**.**

The new command is saved to the MyStuff.LSP file.

#### **5. In the Visual LISP Editor, click **Tools**→**Load Text in Editor**.**

The custom command is loaded into AutoCAD and is now ready for use.

**6. Switch back to AutoCAD.**

**7. At the command line, type BCIRC-AX and press Enter.**

The new command works just like the old one did, except this one isn't dependent on any of the AutoCAD commands, which is why the code is much longer. This code handled passing the point into the AddCircle command. Also, notice how much more complex the layer creation process is compared to before.

