

## MAKING THE MOST OF VISUAL LISP'S NEW FEATURES

---

This paper was developed by Perceptual Engineering, Inc. and presented at Autodesk Design World in Philadelphia on Sunday, September 13, 1998, as part of the "Making the Most of Visual LISP's New Features" presentation.

This paper, as well as other Visual LISP materials and information, are available from Perceptual Engineering's website at [www.perceptual-eng.com](http://www.perceptual-eng.com).

Copyright 1998, Perceptual Engineering, Inc.

## TABLE OF CONTENTS

---

Making the Most of Visual LISP's New Features .....	1
Table of Contents .....	1
Course Description .....	2
PRG-S2    Sunday 12:45-2:15 pm .....	2
Introduction to Visual LISP.....	2
Visual LISP .....	2
Visual LISP Features .....	3
The Visual LISP Interface.....	3
The Console Window – the Heart of Visual LISP .....	4
Visual LISP Editor Windows .....	5
Syntactical Coloring.....	5
Auto-Formatting.....	5
Loading Code and Running a program .....	6
Extra Editing Commands.....	7
Automatic Word Completion.....	8
Complete Word by Match .....	8
Complete Word by Apropos .....	8
Getting Help.....	8
Commenting Styles .....	8
Matching Parentheses.....	9
Debugging with Visual LISP.....	9
The Code Checker.....	9
Setting a Breakpoint, Stepping Through Code .....	10
Using the Watch Window .....	10
Jump to Last Break Source and the Error Trace .....	11
Using ActiveX within Visual LISP .....	12
Why use ActiveX? .....	12
ActiveX in Visual LISP – Basics.....	12
Function Syntax.....	12
The AutoCAD Object Tree .....	13
Accessing the AutoCAD Application Object .....	14
Accessing Other ActiveX Objects within the Tree .....	15

Visual LISP Projects .....	15
Creating a Project.....	16
Overview of the functions available within the project window .....	16
Compiling Project Files .....	18
Visual LISP's Run-Time System .....	18
Creating Visual LISP Applications.....	19
Reactors.....	20
Conclusion .....	20

## COURSE DESCRIPTION

---

### ***PRG-S2      Sunday 12:45-2:15 pm***

Visual LISP has pumped new life into the AutoLISP programming language. The new capabilities it offers promise to make AutoLISP once again one of the most powerful ways of customizing AutoCAD software. In this session, you will learn some of the features available to you, including reactors, debugging, ActiveX, ObjectARX compilation, and more. If you have some AutoLISP programming experience under your belt and you want to take it to the next level, join this session and see what Visual LISP can do for you!

## INTRODUCTION TO VISUAL LISP

---

### ***Visual LISP***

Visual LISP is a professional development tool which offers AutoLISP programmers many of the features available in modern Integrated Development Environments (IDEs). With its introduction, Autodesk acknowledges the millions of lines of AutoLISP code in existence, and assures that they will be even more valuable in the future. Visual LISP consists of several key components:

- ◆ A professional IDE (Integrated Development Environment) designed specifically for the idiosyncracies of the LISP and, more specifically, the AutoLISP language.
- ◆ An AutoLISP emulation capability which means that you can run programs entirely within Visual LISP with nearly complete compatibility.
- ◆ New features and functions that expand the AutoLISP language, including access to ActiveX functionality, and reactor capabilities.
- ◆ The ability to compile AutoLISP code into a secure and fast-loading form of P-Code. In addition, AutoLISP programmers now have the option of producing standalone ARX executable modules, and can deliver applications in a single file, rather than multiple AutoLISP and dialog (DCL) files.

Visual LISP (VLISP) is AutoCAD's own development environment for the creation, debugging, delivery and maintenance of AutoLISP programs. It is itself an AutoCAD program (and an ObjectARX application), so you must have AutoCAD running before you can launch Visual LISP. This relationship of AutoCAD and Visual LISP makes it possible for you to test code while still in the development environment, and to test and examine code while it is actually executing within AutoCAD.

## ***Visual LISP Features***

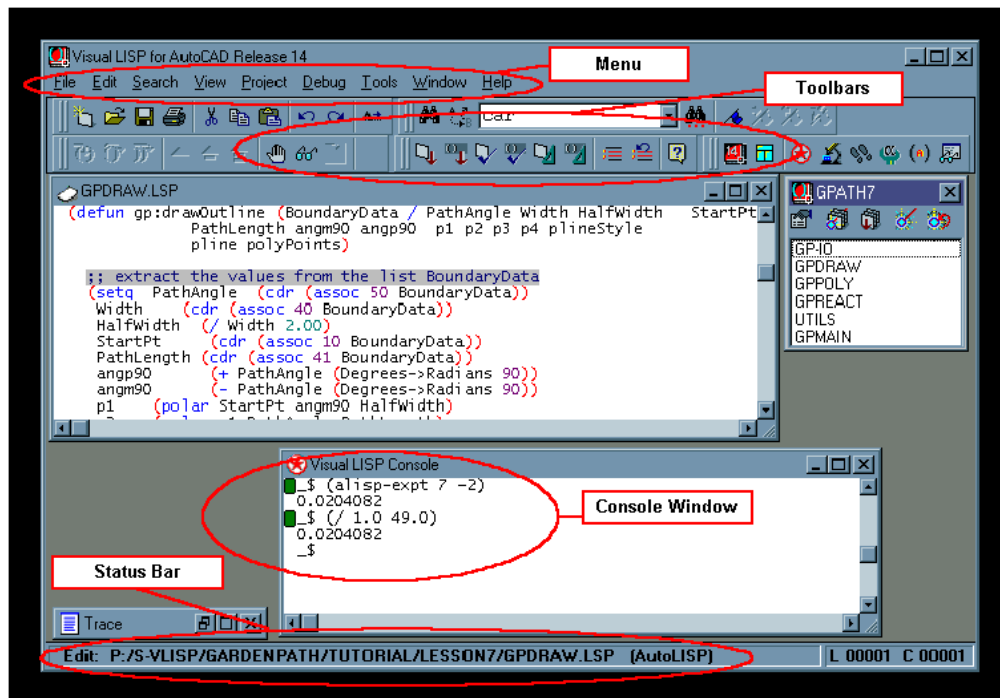
Visual LISP is an integrated development environment, requiring no additional software tools and providing some capabilities previously unavailable even with such tools. It is made up of several functional components, including:

- ◆ Syntax Checker
- ◆ File Compiler
- ◆ Native AutoLISP emulation
- ◆ Debugging utilities designed specifically for AutoLISP code
- ◆ A built-in, customizable AutoLISP formatter
- ◆ Comprehensive Inspector and Watch facilities that provide convenient access to variable and expression values for data structure browsing and modification. These features may be used to explore AutoLISP data and AutoCAD drawing entities.
- ◆ Context sensitive help
- ◆ A Project File system that makes it easy to maintain multiple-file applications
- ◆ Compilation of AutoLISP source code into binary ObjectARX-format executables
- ◆ Desktop Save/Restore capabilities
- ◆ An intelligent System Console whose basic functions correspond to the AutoCAD Text Screen functions and provide a number of interactive features such as history scrolling and full-input line editing

## ***The Visual LISP Interface***

First, you should be familiar with the Visual LISP User Interface. Elements include:

- ◆ **Menu.** Issue VLISP commands by selecting from the menu items. A brief description of the command's function will be displayed in the status bar at the bottom of the screen. Menu contents will vary depending on the current task.
- ◆ **Toolbars.** Toolbar buttons provide a quick way to issue VLISP commands. There are 5 toolbars (Debug, Edit, Find, Inspect and Run) covering most, but not all of the commands available from the menu. Toolbars are detachable – that is, you can move them from the icon ribbon to any location on the screen. However, a few commands are not available from toolbars in the “undocked” state.
- ◆ **Console Window.** This is a separate, scrollable window within the main Visual LISP window. You can type AutoLISP commands in the console window, or use it to issue Visual LISP commands instead of the menu or toolbars.
- ◆ **Status Bar.** The information displayed here varies according to your current activity.



## ***The Console Window – the Heart of Visual LISP***

Like the AutoCAD command window, the Visual LISP Console window is used to enter and run AutoLISP commands, and to see the results of those commands. The Console is also where Visual LISP displays diagnostic messages and the results of AutoLISP functions that are evaluated at the “top level.” You can scroll through the window to view previously entered text and output.

The Console Window may not look like much, but you will be spending a lot of time there as you develop programs – testing code, retrieving values from AutoCAD, examining the results of your program, etc. This is actually one of the most powerful features of Visual LISP.

Here is a brief summary of Console functions:

- ◆ Enter an AutoLISP expression for Visual LISP to read, evaluate and display the results.
- ◆ Enter more than one expression before pressing the ENTER key.
- ◆ Copy and transfer text between the Console and the text editor. Most text editor commands are also available in the Console.
- ◆ Retrieve the previously entered command by pressing the TAB key. Pressing TAB repeatedly retrieves earlier commands. Pressing SHIFT+TAB retrieves commands in the opposite direction.
- ◆ Perform an associative search in the input history.
- ◆ Clear any text following the Console prompt by pressing ESC.
- ◆ Pressing SHIFT+ESC leaves the text you typed at the Console prompt without interpreting it, and displays a new Console prompt. Hitting TAB then brings back the first line and Visual LISP performs the evaluation.

- ◆ Clicking the right mouse button, or pressing SHIFT+F10 anywhere in the Console window, displays a menu of Visual LISP commands and options.

## ***Visual LISP Editor Windows***

Visual LISP provides you with a robust AutoLISP code-editing environment. Any existing AutoLISP files can be opened within the Visual LISP IDE, and new files can be created.

When you create a new editor window by selecting File->New, and then type in AutoLISP code, you should notice two things right away:

- ◆ Automatic indenting is applied as the code is typed in.
- ◆ Syntactical coloring is also applied.

The syntactical coloring is a feature that is applied to all AutoLISP source code – not just new code that you type in using Visual LISP.

## **Syntactical Coloring**



As soon as you type text in the text editor window, VLISP determines if the entered word is a built-in AutoLISP function, a number, a string, or some other known language element. The default color scheme is:



Built-in functions and protected symbols	Blue
Strings	Magenta
Integers	Green
Real Numbers	Teal
Comments	Magenta, on gray background
Parentheses	Red
Unrecognized items	Black

However, you can change the color scheme for each type of file for which Visual LISP provides color coding (LSP, DCL, SQL, C language). VLISP reads the file name extension and provides the appropriate color scheme.

## **Auto-Formatting**

Until Visual LISP, there was no widely-used tool for creating AutoLISP code, so programmers have been pretty much on their own to come up with a readable format.

To format all the text in an active editor window, choose Tools  Format AutoLISP in Editor  from the Visual LISP menu, or click the Format Edit window button on the Tools toolbar.

To format a *part* of your code, select the portion you want to format and choose Tools  Format AutoLISP in Selection  from the Tools menu, or click the Format Selection button on the Tools toolbar.

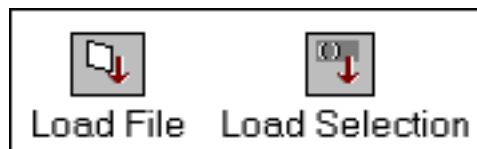


The Formatter issues an error diagnostic if the selected text is not a sequence of valid AutoLISP expressions.

The default style that Visual LISP uses may not be to everyone's taste. You can customize it to suit your own preferences using the Tools Environment Options AutoLISP Format Options menu selection.

## ***Loading Code and Running a program***

The code in an editor window is not active until it has been loaded to the Visual LISP environment. This is similar to the way you used to write code using Notepad (or some other editor), then loaded the program from the AutoCAD command line or through the Appload dialog. The process is much simpler using Visual LISP. Two buttons on the toolbar let you load an entire file, or whatever text is currently selected.



Any C:-defined functions are automatically exported to AutoCAD, so you can invoke your LISP functions either from Visual LISP or from the AutoCAD command line, just as you have always done.

There are other significant differences between working with Visual LISP and the old AutoCAD methods. For example, to examine a variable in Visual LISP, you do not need to precede the variable with an exclamation point like you do in AutoCAD.

One of the benefits of Visual LISP is that it knows what it is looking for and will frequently tell you if you haven't provided it. When working in the Console Window, for example, if you forget a parenthesis, Visual LISP will display the missing parenthesis and wait for you to type it in.

You can navigate back and forth between AutoCAD and Visual LISP using standard Windows techniques:

- ◆ Alt+Tab (task switching)
- ◆ TaskBar

Or you can use methods which are specific to Visual LISP. To get from Visual LISP to AutoCAD, use one of these methods:

- ◆ AutoCAD Icon – the AutoCAD icon is the first icon in the View toolbar of Visual LISP



- ◆ Type (acad) in the Console Window

To get from AutoCAD to Visual LISP at any time, type VLIDE at the command line.

NOTE: One advantage of starting a command or function within Visual LISP is that you can use the Console Window "history" feature, allowing you to retrieve previously-entered keystrokes.

## ***Extra Editing Commands***

If you press CTRL+E while in an active Visual LISP text editor window, VLISP will display a list containing the following editor options. Click on the selection to perform the operation on the selected text.

<b>Task</b>	<b>Description</b>	<b>Keyboard Shortcut</b>
Indent Block	Indents the selected block of text by adding a tab to the beginning of each line	TAB
Unindent	Unindents the selected block of text by removing a tab.	SHIFT+TAB
Indent to Current Level	Indents according to context.	CTRL+ALT+TAB
Prefix With	Adds a text string to the beginning of the current line, or to each line in a block of selected lines, after prompting you for the string.	CTRL+E, then X
Append With	Appends a text string to selected lines of text, after prompting you for the string.	CTRL+E, then W
Comment Block	Converts a block of code to comments.	CTRL+E, then C
Uncomment Block	Changes a block of comments to active text.	CTRL+E, then U
Save Block As	Copies selected text to a new file.	CTRL+E, then A
Uppcase	Converts the selected text to all upper case.	CTRL+E, then P
Downcase	Converts the selected text to all lower case.	CTRL+E, then D
Capitalize	Capitalizes the first letter of each word in the selected text.	CTRL+E, then Z
Insert Date	Inserts the current date (default format is MM/DD/YY).	CTRL+E, then R
Insert Time	Inserts the current time (default format is HH:MM:SS)	CTRL+E, then T
Format Date/Time	Change the Date and Time format	CTRL+E, then M
Sort Block	Sort the selected block of code in alphabetical order.	CTRL+E, then S
Insert File	Insert the contents of a text file into the current editor window at the cursor position.	CTRL+E, then F
Delete to EOL	Erases everything from the cursor position to the end of the current line.	CTRL+E, then E
Delete Blanks	Deletes all the blank spaces from the cursor position to the first non-blank character in the line.	CTRL+E, then B

## ***Automatic Word Completion***

Two text editor features allow you to type in part of a word and have Visual LISP complete the word for you. They are Complete Word by Match and Complete Word by Apropos.

### **Complete Word by Match**

Press CTRL+SPACE to invoke this feature. Visual LISP will complete a partially-entered word by matching the entered portion with another word in the same edit window. The feature is case insensitive, and is used to avoid retyping the same word repeatedly in the same session.

Visual LISP keeps track of the function and variable names within an editor window, and this feature will attempt to match to them. However, there are times when you want to match only to Built-in AutoLISP functions, and that's when you would use the next feature.

### **Complete Word by Apropos**

Visual LISP will also complete a partially entered word with a matching symbol name from the Visual LISP symbol table, which lists the elements in an AutoLISP program – the symbols, subroutines and variables referenced by the program. Enter a portion of the word, then press CONTROL+SHIFT+SPACE to invoke the feature.

If Visual LISP finds several matching names, it presents a context menu from which you can select the name you want. If more than 15 names are found, the Apropos dialog box appears.

## ***Getting Help***

A complete Visual Lisp tutorial is accessible by clicking the Help button in the toolbar.

In addition, the Help function in Visual Lisp is context sensitive, designed to take you quickly to the information you need right now. For example, you may be editing some existing code and run across an AutoLISP function name you're unfamiliar with. To get more information on it, double-click the word, then hit the Help button on the toolbar.

As a bonus, any of the code in a help file is "cut-and-pastable" into Visual LISP.

## ***Commenting Styles***

Visual LISP will automatically format your comments using the following, simple rules:

Comments begin with one, two or three semi-colons. The number determines how the comment is formatted:

**One semi-colon.** Visual LISP will place the comment at the end of the current line of code. Use for short comments.

**Two semi-colons.** Visual LISP will indent the comment to the current level.

**Three semi-colons.** Visual LISP will align the comment with the left margin. Use for extensive or particularly important comments.



NOTE: Using the automatic Commenting feature of Visual LISP will automatically add three semi-colons to the selected text, and align your comment with the left margin. The corresponding UnComment function removes the semi-colons.

## ***Matching Parentheses***

Visual LISP's Parenthesis matching feature makes it easy to highlight and check a program.

Start by placing the cursor at the opening parenthesis of the expression you want to check. Next, hit Ctrl+Shift+]. This will find and select (highlight) the text up to the corresponding closing parenthesis. Repeat the procedure to check the next block of code.

That's the official way of doing things, but there is an even easier way. Rather than going through the cumbersome keystrokes, you can simply double click outside an opening parenthesis. Visual LISP will invoke the parenthesis matching feature and highlight (select) the code between the selected parenthesis and its corresponding closing parenthesis.

## **DEBUGGING WITH VISUAL LISP**

---

### ***The Code Checker***

The Visual LISP code checker is an easy-to-use, first line of defense that can help you clean up some obvious errors before you start looking for the tough ones.

To check the syntax of all the text in an editor window, do the following:

Switch to the editor window containing the code you want to check

Choose Tools  Check Editor  from the VLISP menu

Or select this icon from the toolbar



You can also perform a syntax check on a selected piece of code, using the Check Selection item on the Tools menu.



A new window will appear, displaying the error. Also, the code containing the problem is highlighted.

The code checker will only catch "syntactical" errors. These are errors involving the incorrect number of arguments to a built-in function, or the incorrect form of a function. It won't catch logical errors in your program, or the use of incorrect data types (such as passing a string into a function that requires a numerical value). That's where the additional debugging tools come in handy.

## ***Setting a Breakpoint, Stepping Through Code***

Visual LISP provides several ways that you can suspend the execution of a program – by setting breakpoints in code, or by selecting the “Stop Once” or “Break on Error” modes. When execution of a program is suspended, Visual LISP is in a state called the “Break Loop Mode”. During this state, you can examine the code as the program is executing, and modify the value of AutoLISP objects, including variables, symbols, functions, and expressions.

Visual LISP gives you a toolbar of buttons that are used for setting breakpoints and stepping through code. From left to right they are:



- ◆ **Step Into** jumps into a nested expression, if any. If there are no nested expressions, it jumps to the next expression in sequence.
- ◆ **Step Over** looks for the closing parenthesis matching the opening parenthesis where the program is currently paused, and evaluates the expressions in between.
- ◆ **Step Out** searches for the end of the function where the program is currently paused, and evaluates all of the expressions up to that point.
- ◆ **Continue** resumes normal program execution from the breakpoint.
- ◆ **Quit Current Level** terminates the current break loop and returns to a break loop one level up. This may be another break loop or the top level read-eval-print loop.
- ◆ **Reset to Top Level** terminates all currently active break loops and returns to the Console top-level (the top read-eval-print loop).




The remaining selections are used for setting breakpoints and other miscellaneous functions.

## ***Using the Watch Window***

The Visual LISP Watch window allows you to watch the values of variables while the program executes. The content of the Watch window is updated automatically. In other words, if the value of a variable placed in the Watch window is changed, this change will automatically be reflected in the Watch window.

This is an extremely valuable debugging tool. It means that you are no longer required to write print statements within your sourcecode whenever you want to examine the value of a variable.

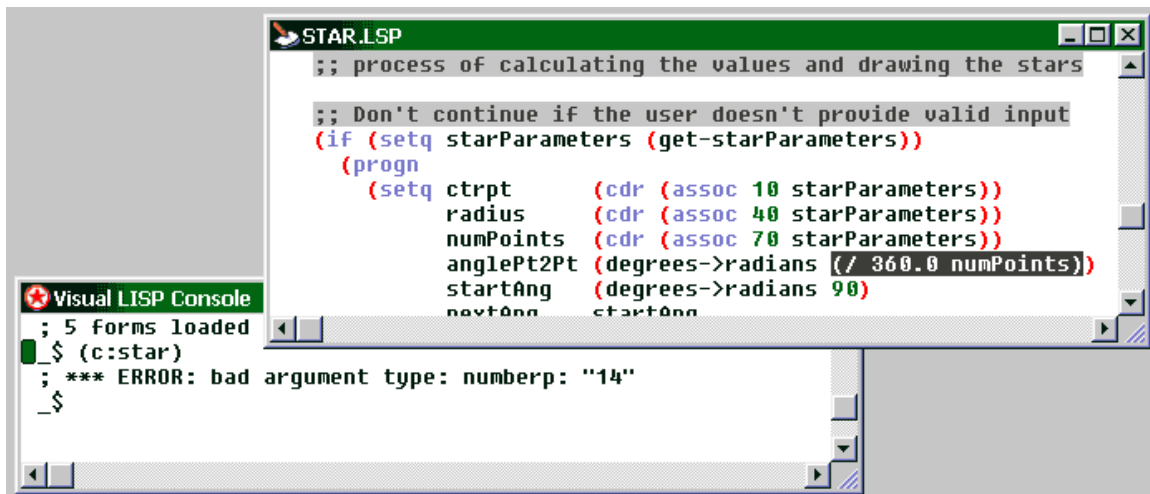
The watch facility can be accessed using one of the following methods:

- ◆ Control+Shift+W
- ◆ Menu bar View  Watch Window 
- ◆ Select the watch button. 

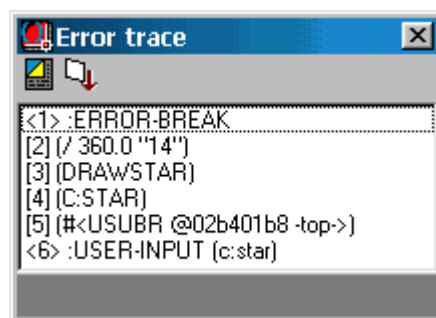
Enter the name of the expression you want to observe in the Expression to Watch window, and click OK.

## ***Jump to Last Break Source and the Error Trace***

In the unlikely event that your program ever crashes (!) you can use two tools to focus in on the location of the problem. The Jump to Last Break Source button can be used to place the cursor right at the location where your code crashed within Visual LISP, and the offending expression is highlighted.



If that doesn't provide enough information, you can use the "View Error Trace" capability to see exactly what parameters were in use when the break occurred.



The program where the break occurred is displayed in the trace window. In this example, you can see that the code was attempting to divide a numerical value (360.0) by a string ("14"). This break occurred within the DrawStar function, which was invoked from the C:Star function.

## USING ACTIVE X WITHIN VISUAL LISP

---

### ***Why use ActiveX?***

One question to consider before expanding your repertoire of LISP functions is “why use ActiveX in the first place?” After all, with all of the other additional LISP functions within Visual LISP, you can already write robust applications. The answers to this question are easy:

- ◆ Speed
- ◆ Increased functionality
- ◆ Code-readability and maintainability

### ***ActiveX in Visual LISP – Basics***

#### **Function Syntax**

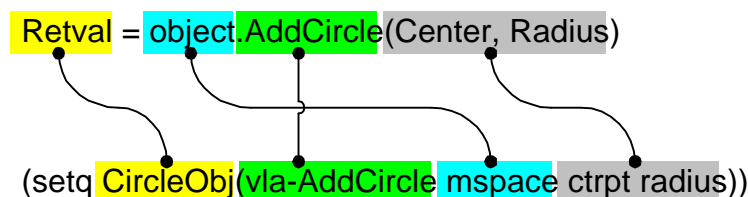
At first, the ActiveX function names within Visual LISP may look a little awkward. But with this “field guide” to ActiveX functions, you can sort through them.

First, there are two major categories of ActiveX functions:

- ◆ Those which correspond directly to built-in AutoCAD ActiveX Methods
- ◆ Those which are specialized Visual LISP functions, unique to the VL environment

Visual LISP functions beginning with “vla-“ fit in the first category, those beginning in “vlax-“ fit into the second.

It is helpful to have a “translation guide” for the “vla-“ functions. There is no native Visual LISP documentation for these functions. Instead, you must use the AutoCAD ActiveX Automation Reference (AAR) Help file. This file provides ActiveX Method syntax, as required for Visual Basic. However, translating this syntax into Visual LISP’s syntax is pretty simple, once you know how:



Here are some tips:

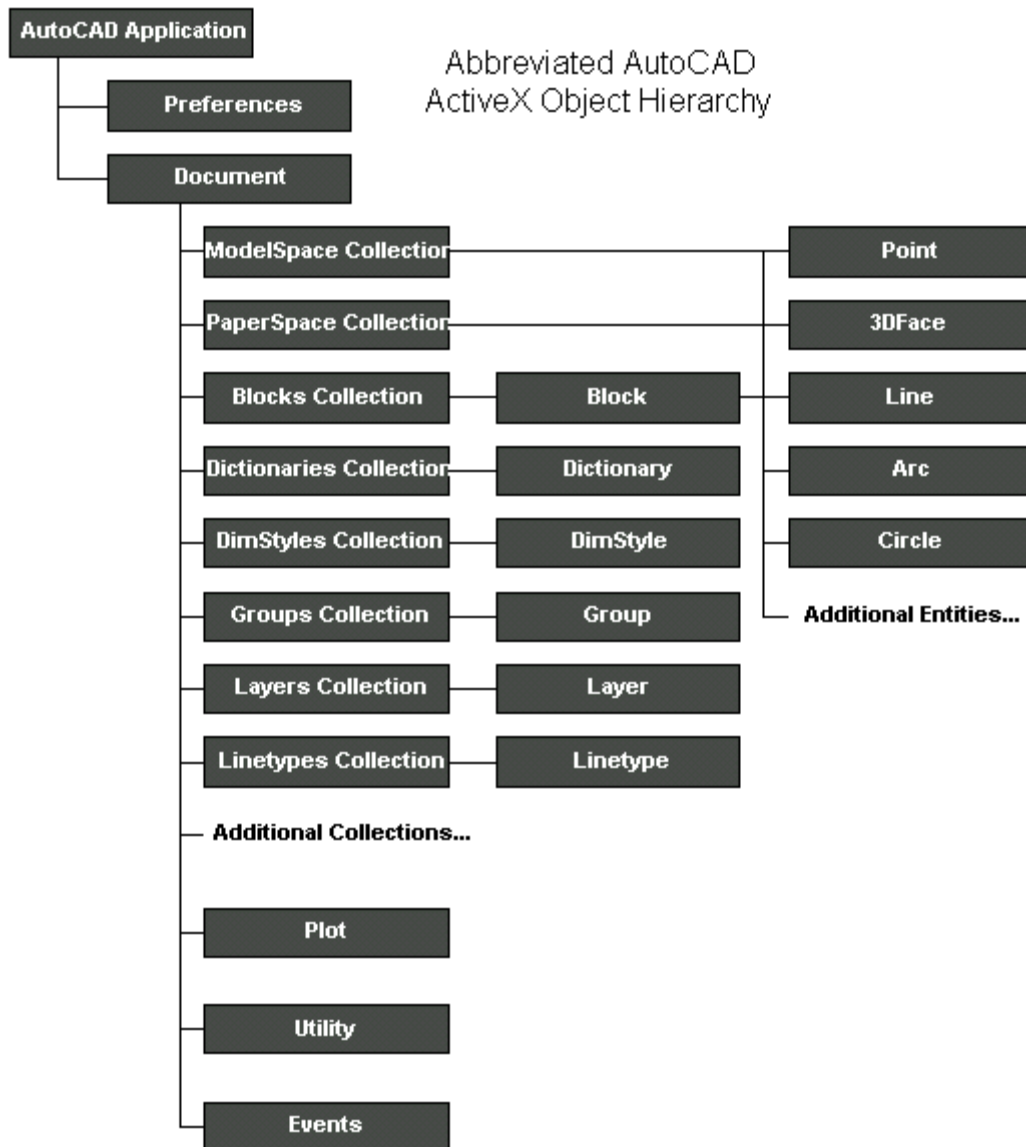
1. Start out your ActiveX function with a `setq`. All ActiveX methods return a value, and most of the time you'll need to use the value. The variable name that you choose will correspond with the “RetVal” in the AAR.
2. Look for the method name. In the above example, “AddCircle” is the method name. Prefix this method name with “vla-“.

3. Determine what kind of object the ActiveX method is related to. This determines the parameter that you must pass to the Visual LISP function. For example, the VL expression shown above uses the variable "mspace," which is assumed to hold a pointer to the model space collection object. (If you don't understand this part yet, hang on – we'll get there soon.)
4. Finally, determine what parameters the ActiveX method is expecting. These must be included in your Visual LISP expression in the same order that they appear in the ActiveX Method reference.

## **The AutoCAD Object Tree**

ActiveX is a standardized interface employed by many Windows applications. An application designed to run as an ActiveX Automation Server (i.e., allowing other programs and applications to automate or run pieces of the server program) must provide an object model to its client programs (i.e., the program that is using the functionality of the server). By understanding a server application's object model, you can determine how to navigate through the functionality the program makes available to the "outside world."

AutoCAD provides such an object model, and it is essential to understand it if you intend to employ ActiveX functions from within Visual LISP. If you are an experienced AutoLISP programmer, you already have an idea of some of the "objects" that AutoCAD exposes through ActiveX. You've probably walked through symbol tables, and accessed entity data. The following illustration provides an overview of the AutoCAD object hierarchy.



NOTE: The AutoCAD Object Hierarchy is illustrated in more detail in the AutoCAD ActiveX Automation Reference, which is accessible through the context-sensitive help for any vla-function.

Every object in the hierarchy has its own behavior and its own information. In ActiveX lingo, these are known as methods (behavior) and properties (information).

## Accessing the AutoCAD Application Object

ActiveX functions are all related to AutoCAD objects. AutoCAD objects are all related through the AutoCAD Object Tree. To do anything at all with ActiveX objects, you have to be able to get at them through the tree.

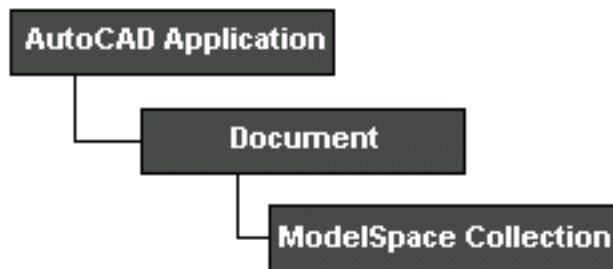
Most of the time, you'll find that you will need pointers to – that is, variables that retain the addresses of – certain key objects. In most cases, you will start out an ActiveX program by

retrieving a pointer to the root object of the AutoCAD Object Tree – which is the AutoCAD Application itself.

```
_S (setq acadApp(vla-get-acad-object))  
#<VLA-OBJECT IAcadApplication 00b3c9b4>  
_S
```

## Accessing Other ActiveX Objects within the Tree

With the object pointer to the AutoCAD Application Object in hand, you can walk through the hierarchy to any other object, using standard “methods.” For example, to get a pointer to model space (which you’ll need in order to add any new entities to your drawing), first examine the object hierarchy and determine how to get to model space, starting at the application object:



The following code should now make sense:

```
_S (setq modelSpace(vla-get-modelSpace(vla-get-ActiveDocument acadApp)))  
#<VLA-OBJECT IAcadModelSpace 021f34c0>  
_S
```

All of the methods and properties of the ActiveX objects are described in the AutoCAD ActiveX Automation Reference. You’ll find that help file nearly as handy as the Visual LISP Help when you’re programming with Visual LISP’s ActiveX capabilities.

## VISUAL LISP PROJECTS

---

A well-designed AutoLISP application may consist of several files, dividing the functions into LISP files which are organized by purpose or type. When you are building a new application, you can simply include these files, along with another new LISP file that contains the code specific to the new application.

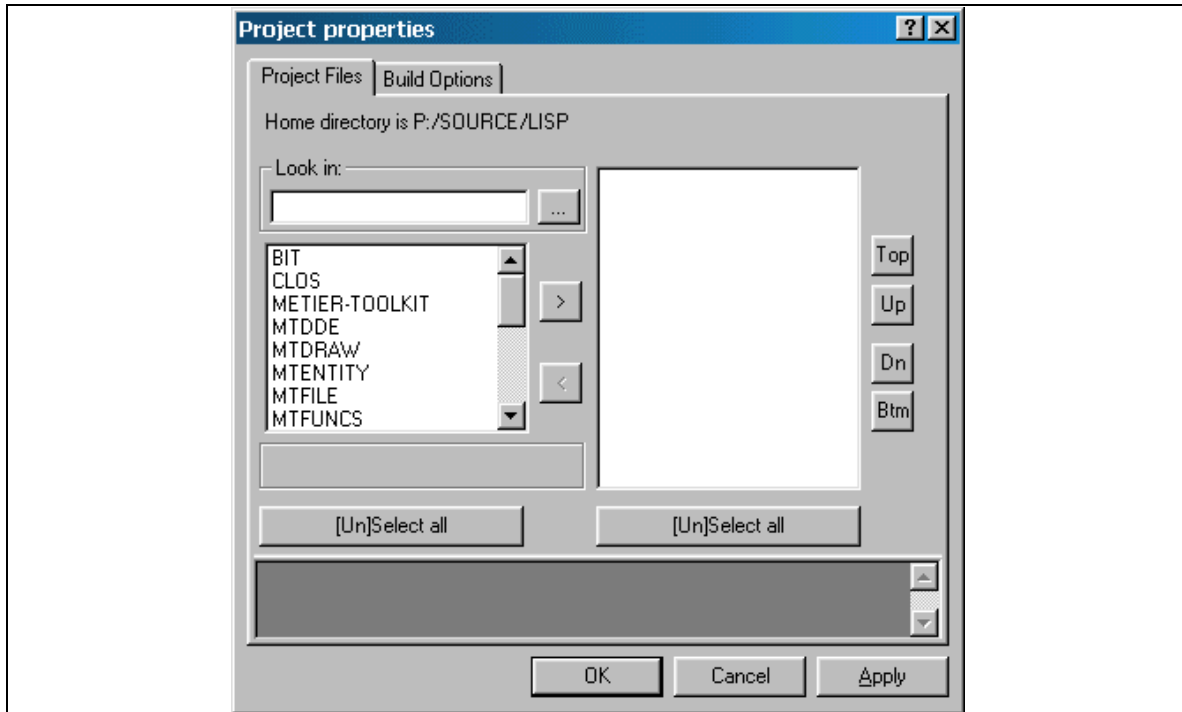
There are several good reasons for using this modular approach to building an application:

- ◆ Files are easier to maintain (it’s easier to find and work with code).
- ◆ You have the ability to re-use existing code in a more efficient way.
- ◆ Smaller files are easier to debug, since you don’t have as much code to worry about.
- ◆ It’s easier to find your way around a large application when it’s broken up into smaller chunks.

To help you work with these multiple-file applications, Visual LISP uses a construct called a “project” – essentially a list of source files and a number of functions that provide functionality for maintaining the project (adding, removing and organizing the files) and compiling the files into binary format.

## Creating a Project

Selecting New Project from the Visual LISP menu begins a process where you can select multiple LISP source files and include them all in a single project. After giving your new project a name, a dialog box appears where you can select the Lisp files that will belong to your project.



## Overview of the functions available within the project window

Visual LISP provides several tools for building and maintaining projects, many available from the Project Window.



The Visual LISP Project Window displays a list of the project's members, in the order in which they will load.

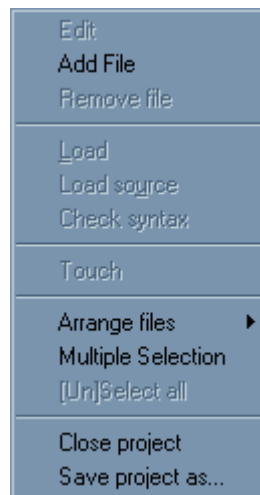
The project name appears in the title bar. Below the title bar are five icons with the following functions:





- ◆ **Project Properties** – Displays the Project Properties window for the project. This allows you to view the full path name of the files in the project, add, remove, and reorder project files, and view and change project compiler options.
- ◆ **Load Project FAS** – Loads all compiled (FAS) files for the project.
- ◆ **Load Source Files** – Loads all the project's source files, making them available to be run.
- ◆ **Build Project FAS** – Compiles all project source files that have been modified since their last compile.
- ◆ **Rebuild Project FAS** – Recompiles all project source files, whether or not they have changed since their last compile.

Right clicking within the file list of the Project Window displays a context menu from which the following functions are accessible:



- ◆ **Edit** – Edit the source code of the selected project members
- ◆ **Add File** – Open the Project Properties dialog, in order to add files to the project
- ◆ **Remove File** – Remove the selected members from the project.
- ◆ **Load** – Load the FAS file for the selected project members. If no FAS file exists for a member, load the LSP file.
- ◆ **Load Source** – Load the source LSP file for the selected project members.
- ◆ **Check Syntax** – Check AutoLISP syntax for the source code for the selected members

- ◆ **Touch** – Indicate that the selected source files have been updated, but make no change to the files. This causes Visual LISP to recompile these programs the next time you ask to compile all changed project files.
- ◆ **Arrange Files** – Sort the project member list, according to one of the available suboptions (load order, name, type or date)
- ◆ **Multiple Selection** – tells Visual LISP whether or not to allow selection of multiple members from the list in the Project Window. If this option is checked, multiple selection is allowed.
- ◆ **Unselect all** – Selects all; members of the project list, if none are currently selected. If any members are currently selected, this command unselects them.
- ◆ **Close Project** – Close the project.
- ◆ **Save Project As** – Save (and name) the project.

## Compiling Project Files

When you run AutoLISP code from the Visual LISP Console, or load code from the Visual LISP text editor and run it, Visual LISP translates your source code into machine-executable code. This is done by the immediate-execution compiler, which plays the same role as the AutoLISP interpreter in AutoCAD.

But Visual LISP has another compiler as well, called the File Compiler, which generates executable machine code from your source files and saves it in files identified by an .FAS extension. This executable code can be run from within Visual LISP, and runs faster than code generated by the immediate-execution compiler.

More importantly, it allows you to create applications that others can use. These can be stand-alone applications, or applications accompanied by the Visual LISP Run-Time Support System (RTS). Compiled executable files contain only machine-readable code, so your source code remains secure no matter how widely it is distributed. Even strings and symbol names are encrypted by the Visual LISP File Compiler.

You can compile files individually, and a number of specialized Visual LISP commands are provided for doing so. However, the project feature in Visual LISP provides compilation options that are simpler to use, and much easier to remember.

One option is to have Visual LISP recompile all source files that have changed since the last time they were compiled. By choosing this option, you insure that all FAS files in your application correspond to the latest versions of the program source code. You also save time by avoiding unnecessary compiles. To invoke this feature, click the Build Project FAS button in the Project window.

You can also choose to recompile all the programs in your project, whether or not they have changed. This command is selected by clicking the Rebuild Project FAS button.

## Visual LISP's Run-Time System

Visual LISP's Run-Time System (RTS) is an ObjectARX program that loads and executes compiled or uncompiled AutoLISP files. It is a specialized application that knows how to run any of Visual LISP's code files, and is an essential component in delivering applications to end-users if you wish to distribute your application in a compiled format.

There are two types of RTS modules, one for applications that reference ActiveX functions and one for applications that do not. Specify the RTS that your application requires.

The ActiveX runtime engine is called "vlarts.arx". The standard runtime engine is named "vlrts.arx".

## CREATING VISUAL LISP APPLICATIONS

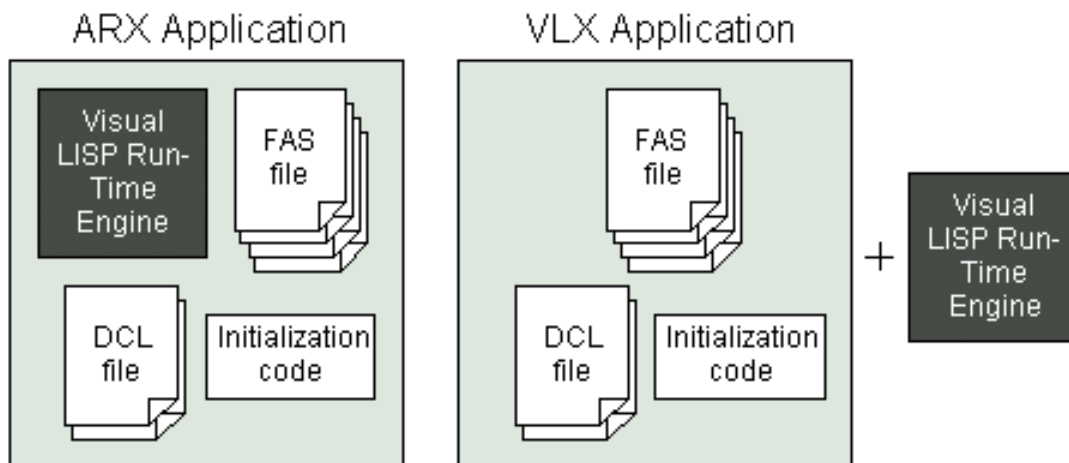
---

An Application is typically, though not necessarily, related to a project. An application is a packaged set of compiled Visual LISP code, delivered along with its own run-time engine, that is compiled into an ARX application.

As a Visual LISP programmer, you can now claim that you can output ARX applications. But you don't need to worry about understanding any of the complexities of writing ObjectARX code. You also shouldn't claim that you're "writing ObjectARX code!" Visual LISP does all of the work for you.

When you want to create an application, there is one very important decision that you must make right at the beginning, and that is the relationship of your compiled code to the Visual LISP Run-Time Engine. This engine is the true ObjectARX component of a Visual LISP application. It is the piece that takes the compiled p-code and turns it into a working, AutoCAD application.

You can either bundle the engine with your code, resulting in an ARX application. Or you can leave the engine on the outside, where it must be loaded separately, in which case you produce a VLX application.



The easiest way to build an application is to use the Visual LISP Application "Wizard," which guides you through the process of creating a compiled, ObjectARX application. One by-product of the process is a Make file (file extension: MKP), which contains all the instructions Visual LISP needs to build the application executable (compile your application's programs, export its functions to AutoCAD, and generate an initialization file to be run in the AutoCAD environment).

**NOTE:** You should build your application only after you have fully debugged it. Compiler errors during the Make process may prevent the Application Wizard from completing successfully.

## REACTORS

---

Reactors allow your application to be notified by AutoCAD when specified events take place. For example, if a user moves an entity to another layer (and there is a properly-designed reactor attached to it), your application will receive notification that the entity has moved. Additional operations can then be triggered.

In effect, attaching a reactor to an entity, or to a drawing editor event, is like protecting your application with a sophisticated alarm system. When an “intruder” performs a certain specified action, the “security firm” (in this case AutoCAD) is informed, and it dispatches the “enforcers” to either prevent the action or enforce the performance of related actions to maintain process standards.

If the reactor object is the alarm, the enforcer is an AutoLISP function that is called by the reactor. Such a function is known as a “callback” function.

There are four types of reactors, corresponding to the general categories of events to which your application can respond:

- ◆ Editor Reactors – An editor reactor will notify you each time an AutoCAD command is invoked.
- ◆ Database Reactors – These reactors correspond to specific entities or objects within a drawing database.
- ◆ Object Reactors – An object reactor will notify you if specific actions are performed to an entity or object within AutoCAD.
- ◆ Linker Reactors – These reactors notify your application every time an ARX application is loaded or unloaded.

Within these general categories, there are many specific events to which you can attach a reactor. When you are designing a reactor-based application, it is up to you to determine the actions that you’re interested in. Once you have done this, you can attach your reactor to the event, then write the callback function that is triggered when the event occurs.

Reactors are a very complicated subject, and require much more description than we can provide here. PEI has produced a separate document with sample code that allows you to run through a few exercises and get a feel for the development of reactor-based applications. To gain a real working understanding of the subject, however, we recommend that you attend a full Visual LISP training session provided by someone with both an overall knowledge of AutoCAD and Visual LISP, and a good understanding of advanced programming techniques using ActiveX and reactors.

For a free copy of PEI’s Reactor Overview document, along with other Visual LISP information and links to other Visual LISP sites, visit us at [www.perceptual-eng.com](http://www.perceptual-eng.com). Or try Autodesk’s own Visual LISP home page at <http://www.autodesk.com/products/acadr14/compapps/vlisp.htm>.

## CONCLUSION

---

We have provided just a brief overview of Visual LISP, but it should be enough to demonstrate what a powerful and complex environment this is.

Visual LISP is truly the future of AutoLISP. By its introduction, Autodesk has signalled that it recognizes the value of the millions of lines of AutoLISP code currently hard at work in offices

throughout the world. Better yet, they have made that code even more valuable by making it more flexible and easier to work with.