The Visual LISP Developers Bible 2003 Edition

By David M. Stein

Visual LISP Development with AutoCAD 2004

Release Log

May 31, 2002 –	Initial public release after having no offers to publish for profit. Bummer.
July 6, 2002 –	Second public release. Added more to the chapter on Safearrays and Variants, minor fixes throughout book.
August 5, 2002 –	Third public release. Added more to the chapter on Reactors.
September 10, 2002 –	Minor updates and corrections only.
October 15, 2002 –	Minor corrections and formatting changes. Updated acknowledgements section.
December 24, 2002 –	Added more information on ObjectDBX and Xrecords.
March 5, 2003 –	Updated to include information about changes introduced by AutoCAD 2004 in chapter 25 (new chapter).

Acknowledgements

I would like to personally thank the following people for their help and guidance throughout the making of this book: My Wife Kathy, my kids, Brad Hamilton, Jon Szewczak, Phillip Ash, my brothers Larry and Joel, my sister Majel, and or course, God.

I would like to thank the following people for their insight and wisdom in general, which has no doubt led me to undertake something like this: Joe Sutphin, Frank Zander, Frank Moore, Jerry Milana, Andrew Hancock, Bud Schroeder, Tom Nelson, Mike Weksler, Kenny Ramage, Frank Oquendo, Bill Kramer, Owen Wengard, Rheini Urban, Randall Rath, Brett Rivers, Bob Leider, Joel Screbnick and Rob Spitzer. I would finally like to pay homage to the following for have enlightened my programming skills by exposing me to some of the most elegant and awesome coding and logic I've seen anywhere: Andrew Hancock, Brad Hamilton, Randall Rath, Randy Kintzley, and Joel Screbnick.

I would like to thank my late parents for bringing me here.

I would like to thank the following for helping me by reviewing the manuscript for this book and providing excellent comments and corrections: Kenny Ramage, Sherko Sharif, Jon Szewczak, Phillip Ash, and many others I can't hope to name.

Persons mentioned above, or anywhere within this document, that may work for certain known companies are named out of appreciation for their generosity, renown expertise and overall compassion for helping others to better understand and leverage the software technologies discussed herein. It is in no way a statement of acknowledgement, approval or condonement on behalf of their employers or themselves in any manner. Some of these people do not even know I've named them herein. Ho ho ho.

Contents

About the Author ... 7
Introduction ... 8
  What is Visual LISP? ... 8
  Comments used Throughout This Book ... 9
  The Future? ... 9
Chapter 1 - The Visual LISP Development Environment ... 10
  The Visual LISP IDE Toolbars ... 11
  The VLISP IDE Pull-Down Menus ... 12
Chapter 2 – Basic Coding in Visual LISP ... 14
  Comparing AutoLISP to Visual LISP/ActiveX ... 15
  Exploring Object Properties and Methods ... 16
  ActiveX vs. DXF? ... 18
    Selection Sets ... 18
    Point Lists ... 18
    Entity Properties ... 18
Chapter 3 – Using ActiveX with Visual LISP ... 19
  Classes ... 19
  Objects ... 19
  Class Inheritance ... 20
  Collections and Dictionaries ... 20
  Properties, Methods and Events ... 21
    Property Relevance ... 22
    Using Methods ... 24
  Data Types ... 26
    Constants and Enumerations ... 27
    Variants and Safearrays ... 27
  Namespaces ... 28
  Interfaces and Type Libraries ... 29
Chapter 4 – Debugging Code with Visual LISP ... 32
    Breakpoints ... 32
    Stepping ... 34
    Animation ... 34
    Watches ... 35
    Tracing ... 36
    Inspection ... 36
    Symbol Service ... 37
    Apropos ... 37
    Bookmarks ... 39
    Goto Line Position ... 39
    Error Trapping ... 40
  Visual LISP Error Trapping Functions ... 40
Chapter 5 – Working with Projects and Multiple Files ... 46
Chapter 6 – Working with Variants and Safearrays ... 48
  Visual LISP Variant Functions ... 48
    Variant Data Types ... 49
  Visual LISP SafeArray Functions ... 50
Chapter 7 –Object Manipulation Functions ... 56
Chapter 8 –File and Directory Functions ... 59
Chapter 9 –Mapping and Iteration Functions ... 63
Chapter 10 –Working with Namespaces ... 66
  Namespace Scoping ... 67
  Namespace Functions ... 67
Chapter 11 –Registry Functions ... 74
Chapter 12 – Reactors and Call-Backs ... 77

Visual LISP Reactor Functions ... 78
Reactor Types ... 79
Verifying Reactor Types ... 82
Using Object Reactors ... 84
Attaching Data to Reactor Objects ... 86
Inspecting Reactors Within the VLIDE ... 86
Querying Reactors ... 87
Transient and Persistent Reactors ... 88
Opening Drawings with Persistent Reactors ... 88
Reactors and Multiple Namespaces ... 88
Guidelines for Using Reactors ... 89
Chapter 13 – Making Visual LISP Applications ... 92
Why Make VLX Applications? ... 92
Building a Simple Application ... 92
PRV Files ... 97
Chapter 14 – Using ObjectDBX with Visual LISP ... 98
What is ObjectDBX? ... 98
How to Use ObjectDBX within Visual LISP ... 98
Chapter 15 – XDATA and XRECORDs ... 103
Working with XDATA ... 103
Working with XRECORD Objects ... 103
Chapter 16– The AutoCAD Application Object ... 107
Chapter 17 – AutoCAD Entities ... 111
All Objects – Common Properties ... 111
The ARC object ... 111
The CIRCLE entity ... 112
The RotatedDimension (LinearDimension) entity ... 112
The ELLIPSE entity ... 113
The LEADER entity ... 114
The LINE entity ... 114
The LWPOLYLINE entity ... 114
The MLINE entity ... 114
The MTEXT entity ... 114
The POINT entity ... 115
The POLYLINE entity ... 115
The RAY entity ... 115
The SOLID entity ... 116
The SPLINE entity ... 116
The TEXT entity ... 116
The TRACE entity ... 116
The VIEWPORT entity ... 116
The XLINE entity ... 116
Chapter 18 – Documents ... 117
The Documents Collection ... 117
Chapter 19 – The Preferences Objects ... 121
The AcadPreferences Collection Object ... 121
The DatabasePreferences Object ... 124
Reloading a Profile ... 125
Chapter 20 – Menus and Toolbars ... 128
The MenuBar Object ... 128
Getting MenuBar Items ... 129
Inserting PopMenus into the MenuBar collection ... 129
Removing PopMenus from the MenuBar collection ... 130
The MenuGroups Collection Object ... 130
The MenuGroup Object ... 130
The PopMenus Object ... 131

The PopMenu Object 131
The Toolbars Collection Object 131
The Toolbar Object 131
Creating a Toolbar 133
Chapter 21 – Interfacing with Other Applications 135
Microsoft Excel 135
Windows Scripting Host 137
The FileSystem Object 138
Windows Messaging and CDONTS 139
Windows Management Instrumentation (WMI) 141
Working with Services 142
Chapter 22 –Using Visual Basic DLLs with Visual LISP 143
Registering DLLs 146
Re-Registering DLLs 146
Chapter 23 – Working With Dialog Forms 148
Referencing DCL Definitions 148
Dynamic Dialog Interaction 149
Controlling Images From Call-Backs 150
Chapter 24 – Examples of Common Tasks 153
Example 1 – Dumping a List of Layer Properties 153
Example 2 – Set All Entities to "ByLayer" 154
Example 3 – Purge, Audit and Save all Opened Drawings 154
Example 4 – Zoom Extents and Save all Opened Drawings 155
Chapter 25 - Changes in AutoCAD 2004 156
General Changes 156
Visual LISP Changes 156
True Color Properties 157
Changes to the ObjectDBX Interface 158
Changes to External Referencing of AcadApplication 159
Changes to AcadPreferences 159
System Variables 162
New Commands 163
Modified Commands (Since AutoCAD 2002) 164
Tool Palettes 164
Conclusion 167
Appendix A - VLAX Enumeration Constants 168
Appendix B – VLISP IDE Keyboard Shortcuts 170
Appendix C – Tips & Tricks for Visual LISP 171
Adding VLX support to the (autoload) function 171
Saving your VLIDE configuration settings 171
Recovering DCL Code from VLX Files 171
Using Projects and DCL with the Make Application Wizard 171
Team-based VLX Development 172
Appendix D – Useful Resources 173
Glossary 174

About the Author

David Stein was born. He lives in Virginia, which is in the United States. For those outside of the US, Virginia is south of New York, north of Florida and east of California. Why am I writing in third person? The particular area I live is very military oriented. All branches of the military are heavily represented here, but being that we're on the coast, the Navy and Coast Guard are the most prevalent.

After 15 years of working as a drafter and designer in the shipbuilding industry, I graduated in 1999 from a local university with a Bachelor's of Science in Information Science and now work as an IT Manager at a large defense contractor. My job title is Manager of Software Applications, but I evolved into this from my previous role as Manager of Engineering and Design Applications. I have been writing various types of program code for over ten years, but have always been involved in AutoCAD programming as a continuum of sorts. The story behind this is so mundanely trivial and unexciting that to commit it to writing should be a punishable crime.

Today, I split my time between CAD applications development, web-based applications development, network administration, strategic business development, washing dishes, taking out the trash, taking my four kids to the beach every day, and drinking beer. I also develop software for routine network administration tasks using many different toolsets.

In all, I work with languages like Visual Basic, Visual InterDev, FrontPage, Microsoft Access, VBA, AutoLISP, Visual LISP, Windows Scripting Host (WSH), Windows Management Instrumentation (WMI), Active Directory Services Interface (ADSI), Cold Fusion, SQL, Active Server Pages (ASP), Kix Scripting, Windows Shell programming (aka Batch and CMD files), Microsoft Systems Management Server (SMS) administration and scripting. I also perform such vital duties as general coffee drinking and bullshit talking to whomever can't run fast enough to escape me. Phew! Do you really care about any of this? Sheesh!

Introduction

This book is aimed at helping experienced AutoLISP programmers better understand and use Visual LISP. This book will cover topics such as ActiveX, Compiling Code, Debugging, Formatting, Deploying and using advanced features such as reactors and namespaces. AutoLISP fundamentals are left for other books to cover as that topic has been aptly covered elsewhere already. This book will focus solely on the Visual LISP extensions to AutoLISP and the unique capabilities and features Visual LISP provides.

For this book, you will need to have access to using AutoCAD 2002 or other Autodesk products that include the Visual LISP toolset such as Mechanical Desktop or Map. Note that Visual LISP is not included with, or usable by products such as Inventor or AutoCAD LT. Nor is Visual LISP provided within competing products such as IntelliCad™ or CadKey®.

What is Visual LISP?

Visual LISP began life as a product of Basis Software, originally named *Vital LISP*. Autodesk purchased the full rights to Vital LISP during the late stages of AutoCAD R14 and renamed it *Visual LISP*. It was then available as a separate add-on to AutoCAD R14. With the release of AutoCAD 2000, Visual LISP replaced the older *Proteus* AutoLISP interpreter module and became an integral part of AutoCAD. It was then incorporated as the LISP interpreter in all AutoCAD-based vertical products, such as Map, Mechanical and Mechanical Desktop.

Visual LISP is more than simply a replacement for AutoLISP, in fact it still works with older AutoLISP code just fine, but it also includes many new improvements. Among the differences are a built-in syntax-aware code editor, dialog previewing, debugging tools, formatting tools, online development references, a compiler and compiler wizard, workspace project management and more.

But the most significant changes to the language itself are due to the addition of ActiveX interface functionality. This effectively puts Visual LISP *potential* on par with other ActiveX technologies such as Visual Basic for Applications (VBA). While Visual LISP still lacks many of the sophisticated tools VBA has, it does possess the capability to interface with ActiveX providers and consumers such as Microsoft Office, Microsoft Windows, and even AutoCAD itself, in ways not possible with AutoLISP alone.

While Autodesk has revised Visual LISP somewhat from its origins in Vital LISP, most of Vital LISP features have not changed much. Visual LISP could be improved to make it an even more powerful development platform, but it seems Autodesk is more interested in other technologies such as VBA, ObjectARX and XML, than ugly old LISP.

Autodesk chose to exclude support for many features in Vital LISP when moving it to Visual LISP. This is unfortunate and unfounded given that the overwhelming majority of development is still done in LISP or Visual LISP. While many Vital LISP features still exist in Visual LISP, their documentation is not available and this makes certain features risky to use, let alone just figure out. If you happen to be an old Vital LISP user, you should be aware of this. Some of these features will be explained later in this book.

Comments used Throughout This Book

Some notations will be shown throughout this book that denote specific types of information. For example…

Tip! will denote information that may not be documented elsewhere or may be difficult to find.

Warning! will denote information that you should be aware of in order to avoid problems or errors in your programming code.

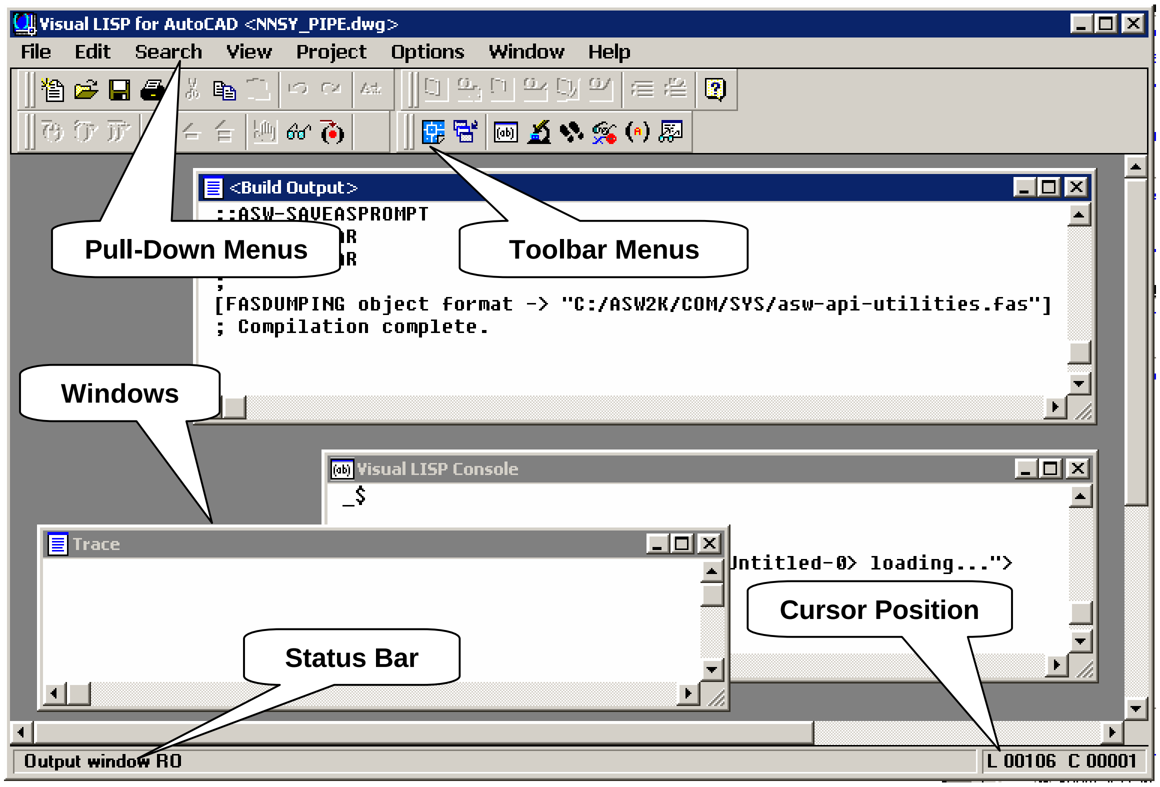
The Future?

Good question. Wouldn't we all want to know the future. As for the future of Visual LISP, it's anybody's guess. Autodesk has provided nothing to indicate what the future of Visual LISP might be. Rumors abound that it will be replaced by VBA, but that seems unlikely in the near term anyway. Also, at the time of this writing, new technologies are emerging such as Microsoft's .NET framework (http://www.microsoft.com/vs), and new variants of LISP such as SharpLISP by 3rd Day Software (http://www.objectdcl.com).

In my humble opinion, Visual LISP is an extremely powerful, flexible and dynamic language to develop upon with respect to CAD applications. It could do much more if given a little nourishment, but it seems the refrigerator is a bit empty lately. Until something comes along that can fully replace it without any limitations, I will continue using it along with the dozen or so other languages I strap on each day to do my work.

Chapter 1 - The Visual LISP Development Environment

The VLISP IDE (Integrated Development Environment) is a combination of tools to help make coding, testing, debugging and compiling output easier and more productive. Type in VLIDE at the AutoCAD command prompt to open the Visual LISP editor. This will load the VLISP ObjectARX application interface (vlide.arx) that loads and enables the IDE for use while AutoCAD is in use. Because Visual LISP is an integral part of AutoCAD, you cannot use the VLISP IDE without AutoCAD being in use.



```
;;ASW-SAVEASPROMPT
R
R
;
[FASDUMPING object format -> "C:/ASW2K/COM/SYS/asw-api-utilities.fas"]
; Compilation complete.
```

```
_$
Untitled-0> loading...">
```

```
Output window RO                                L 00106  C 00001
```

Figure 1-1 – The Visual LISP IDE

Note the IDE features shown in Figure 1-1 above. The top portion includes the IDE pull-down menus and toolbars. The mid-section includes the Build Output, Visual LISP Console and Trace windows. This is also where open program code is shown and edited in their own windows (one per file). Other windows (not shown above) include the Watch window, Object Inspection and APROPOS window.

The bottom edge of the IDE window contains the Status bar. This is where messages are displayed following every action in the IDE. The bottom-right panel is the code editor cursor location display. This shows the current position of the cursor in the code file where *L nnnnn* is the line number and *C nnnnn* is the character offset number. In this example, the cursor is positioned on line 106 on the first character of that line.

The Visual LISP IDE Toolbars

There are four individual VLISP IDE toolbars available. You can move them, dock or undock (float) them as well as hide or show each of them to suit your particular tastes. The toolbars do not exactly match their corresponding pull-down menu features to be careful not to assume that everything exists on a toolbar for being accessed in the VLISP IDE window. You may find that pull-down menus are more effective.

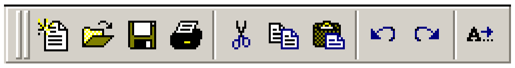


Figure 1-2 – The STANDARD toolbar

The STANDARD toolbar includes general file management features from left to right:

- New
- Open
- Save
- Print
- Cut
- Copy
- Paste
- Undo
- Redo
- Complete Word

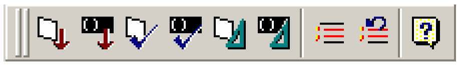


Figure 1-3 – The TOOLS toolbar

The TOOLS toolbar contains general editor features from left to right:

- Load File
- Load Selection
- Check File
- Check Selection
- Format File
- Format Selection
- Comment Selection
- Uncomment Selection
- Help

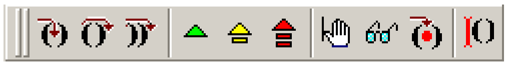


Figure 1-4 – The DEBUG toolbar

The DEBUG toolbar contains tools for testing and debugging code during controlled execution. This includes from left to right:

- Step Into
- Step Over
- Step Out Of
- Continue
- Stop
- Quit
- Toggle Breakpoint
- Add Watch
- APROPOS
- Last Break

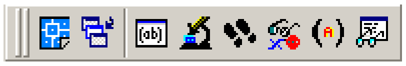


Figure 1-5 – The VIEW toolbar

The VIEW toolbar contains options from left to right:

- Activate AutoCAD (switch to AutoCAD editor)
- Display LISP Console
- Inspect Object
- Trace Window
- Symbol Service
- Apropos
- Display Watch Window

The VLISP IDE Pull-Down Menus

The VLISP IDE pull-down menus are always available by default within the IDE window, whereas the toolbars can be moved, hidden and so forth. As was mentioned in the section above, the pull-down menus contain much more in terms of VLISP editor commands than do the toolbars. For this reason, you may find using the pull-down menus more efficient and effective for daily coding chores.

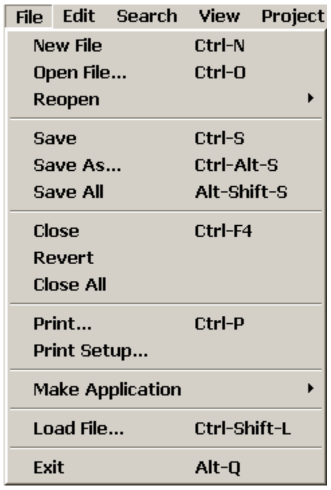


Figure 1-6 - File Pull-down

The File pull-down menu contains standard file management options such as Open, New, Save, Print and Exit. It also provides useful commands like Revert, Close All, Save All and Load File. The Make Application features are discussed later in Chapter 13.

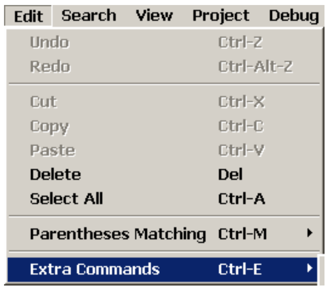


Figure 1-7 – Edit pull-down

The EDIT pull-down menu contains standard clipboard commands, as well as parenthesis matching and a special fly-out named “Extra Commands” that provides some useful editing command features.

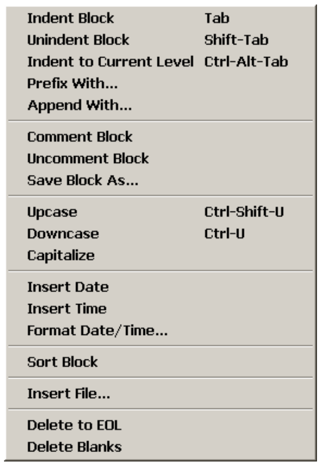


Figure 1-8 – Extra Commands

The Extra Commands fly-out menu, located under the Edit pull-down menu, contains quite a few hidden commands that can save you a lot of time and effort editing code. In particular, Prefix With and Append With, UpCase and Downcase, Capitalize, Insert Date, Time and Sort Block.

..

Chapter 2 – Basic Coding in Visual LISP

In this chapter we will begin writing some basic code using Visual LISP and walking through a simple process for coding, testing, debugging and compiling your code into a finished product. For the sake of trying to at least remain relevant to what a CAD programmer expects, this will not involve the customary "Hello World" coding stuff.

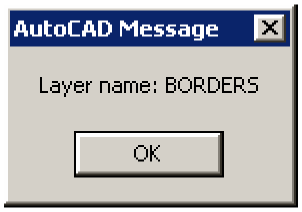
(vl-load-com)

In order to use any of the cool ActiveX functions in Visual LISP, you must first initialize the ActiveX interface by using the (vl-load-com) function. This can be included in every file or every function definition, it doesn't matter. Once it has been executed, subsequent calls do no harm whatsoever.

```
(defun C:SHOWLAYER ( / ent lay)
  (if (setq ent (entsel "\nSelect object to view layer name: "))
    (alert
      (strcat "Layer name: "
        (vla-get-layer (vlax-ename->vla-object (car ent)))
      )
    )
  )
  (princ)
)
```

Figure 2-1 – SHOWLAYERS.LSP

The code in Figure 2-1 demonstrates how to get the layer assignment of a selected entity and display it in a simple alert box. Load this code into AutoCAD and type SHOWLAYER at the command prompt to run it. You will be prompted to select an object on screen "Select object to view layer name: " upon which the object's layer name is then displayed as follows…



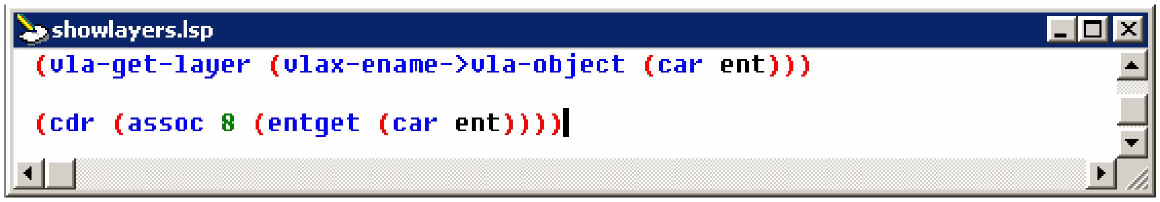
Layer name: BORDERS

While the differences between how you might traditionally access the layer name using DXF entity access is only slight, the user does not need to know that DXF field 8 is the layer assignment. They can instead use (vla-get-layer) which is a bit more intuitive. This is the crux of what makes the ActiveX features in VLISP attractive: clarity.

Tip!

You can add (vl-load-com) to your startup suite in many ways. You can add it to your acad.lsp or acaddoc.lsp file. You can make a small LSP file and select it in APPLOAD as part of your "startup suite".

Comparing AutoLISP to Visual LISP/ActiveX



```
(vla-get-layer (vlax-ename->vla-object (car ent)))

(cdr (assoc 8 (entget (car ent))))
```

Figure 2-2 – Comparison between LISP and Visual LISP methods

Both expressions shown in Figure 2-2 will accomplish the same thing essentially. While the first expression is a bit more wordy, and actually consumes slightly more system resources to execute, the increased baggage is ultimately negligible in most respects.

Once the initial access is made to either root collection of properties (namely, entget or vlax-ename->vla-object), which is normally done once per object manipulation, the rest is actually simpler to write in ActiveX form. For example…

```
(defun GETLAYER (entity / elist)
  (cdr (assoc 8 (entget entity)))
)
```

…is functionally identical to the following…

```
(defun GETLAYER (entity / obj)
  (vla-get-layer (vlax-ename->vla-object entity))
)
```

This is not a comprehensive comparison by any means, since this doesn't demonstrate how the ActiveX object model allows you to navigate relationships in a logical manner. For example, the following code shows how you can retrieve a property setting from the Preferences/Files collection:

```
(vla-get-supportpath
  (vla-get-files
    (vla-get-preferences (vlax-get-acad-object))
  )
)
```

The above capability is not possible to accomplish with AutoLISP alone. It is made possible by ActiveX and the object model of AutoCAD, and the fact that Visual LISP and VBA can access these features through their ActiveX interface to AutoCAD.

Using another example of accessing a particular LINE entity's properties, you can see how the ActiveX interfaces provide very easy to understand names that make coding more intuitive:

```
(setq ent (car (entsel "\nSelect line object: ")))
(setq objLine (vlax-ename->vla-object ent))
(vla-get-layer objLine)
(vla-get-color objLine)
(vla-get-lineweight objLine)
(vla-put-layer objLine "0")
```

```
(vla-put-color objLine acRed)
```

As you can see from this example, it is much more intuitive to access and modify entity properties through ActiveX than by using the more cryptic DXF code numbers. Also, it is worth noting that while the DXF 62 code is transient, the Color property of an entity is persistent. To put this yet another way: An entity that has color=ByLayer has no DXF 62 field in the (entget) data list. Only when a color is applied to override the layer defaults will the entity have a DXF 62 field. However, if you access the same entity through ActiveX, even with color=ByLayer, the return value will be acByLayer.

As an example of how this might be of use to you as the developer, consider the following function that copies layer, color and linetype properties from one entity to another:

```
(defun CopyLayerColor1 (obj1 obj2)
  (vla-put-layer obj2 (vla-get-layer obj1))
  (vla-put-color obj2 (vla-get-color obj1))
)
```

You'll notice that we don't have to rely upon DXF codes, nor do we need to use (subst) or (entmod) functions to update the entity properties. This same function written in AutoLISP might look something like the following example:

```
(defun CopyLayerColor2 (ent1 ent2 / elist1 elist2 lay1 col1)
  (setq elist1 (entget ent1)
        elist2 (entget ent2)
        lay1   (cdr (assoc 8 elist1))
  )
  (setq elist2 (subst (cons 8 lay1) (assoc 8 elist2) elist2))
  (if (assoc 62 elist1)
    (progn
      (setq col1 (cdr (assoc 62 elist1)))
      (if (assoc 62 elist2)
        (setq elist2 (cons (cons 62 col1) elist2))
        (setq elist2 (subst (cons 62 col1) (assoc 62 elist2) elist2))
      )
    )
  )
  (entmod elist2)
)
```

Notice the additional checking required for the DXF 62 code existence in both the source and target entity data lists. As you can see, Visual LISP and ActiveX can dramatically reduce the amount of code required to perform many common tasks. Reducing code also reduces the potential for errors. Reducing errors also reduces the amount of effort required to test and debug code. All of this results in faster, easier and more productive programming with better quality results. And this makes for happier customers!

Exploring Object Properties and Methods

If you haven't already discovered the various properties and methods provided by objects within AutoCAD, a very good way to start is by getting familiar with the `(vlax-dump-`

`object)` function. This function requires one argument, the *object*, to request the object properties, and an optional argument, a flag (anything non-*nil*) to request the object methods.

(vlax-dump-object *object [show-methods]*)

Example of using this function on a standard LINE entity as follows:

```
_$ (setq e (entsel)); pick a LINE entity
_$ (setq obj (vlax-ename->vla-object (car e)))
Returns #<VLA-OBJECT IAcadLine 00f20024>

_$ (vlax-dump-object obj T)
; IAcadLine: AutoCAD Line Interface
; Property values:
;   Angle (RO) = 0.630844
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;   Color = 256
;   Delta (RO) = (4.98519 3.64122 0.0)
;   Document (RO) = #<VLA-OBJECT IAcadDocument 00ed1e0c>
;   EndPoint = (9.63516 6.56966 0.0)
;   Handle (RO) = "2B"
;   HasExtensionDictionary (RO) = 0
;   Hyperlinks (RO) = #<VLA-OBJECT IAcadHyperlinks 00f21c44>
;   Layer = "0"
;   Length (RO) = 6.17338
;   Linetype = "ByLayer"
;   LinetypeScale = 1.0
;   Lineweight = -1
;   Normal = (0.0 0.0 1.0)
;   ObjectID (RO) = 1074179416
;   ObjectName (RO) = "AcDbLine"
;   OwnerID (RO) = 1074179320
;   PlotStyleName = "ByLayer"
;   StartPoint = (4.64998 2.92844 0.0)
;   Thickness = 0.0
;   Visible = -1
; Methods supported:
;   ArrayPolar (3)
;   ArrayRectangular (6)
;   Copy ()
;   Delete ()
;   GetBoundingBox (2)
;   GetExtensionDictionary ()
;   GetXData (3)
;   Highlight (1)
;   IntersectWith (2)
;   Mirror (2)
;   Mirror3D (3)
;   Move (2)
;   Offset (1)
;   Rotate (2)
;   Rotate3D (3)
;   ScaleEntity (2)
;   SetXData (2)
;   TransformBy (1)
;   Update ()
```

As you can see, this is a very helpful function for inspecting entities for their properties and methods. It is also helpful for inspecting *any* other objects, including application objects, documents, collections, and so forth.

ActiveX vs. DXF?

Can ActiveX do everything you need in VLISP to handle all your programming chores? No. There are quite a few situations where the older AutoLISP approach is the only solution to a given problem. In other situations, you could use either one but the AutoLISP approach will turn out to be the most efficient or *manageable* choice. You may reply "Oh sure, that's what you think." But let's look at some scenarios.

Selection Sets

You can create and iterate selection sets, or more properly named *picksets*, using either AutoLISP or Visual LISP. However, you will quickly find that dealing with selection sets in AutoLISP is far easier and less problematic than with VLISP.

Point Lists

Actually, any LIST structure is easier to manipulate in AutoLISP than is the case with an array using VLISP. While both are powerful and flexible, constructing and modifying LIST structures in LISP is much simpler than that of ARRAY structures using VLISP.

Entity Properties

While most properties are easier and more intuitive to access with ActiveX using VLISP, some are not exposed and are therefore only available from DXF code values using AutoLISP. For example, control points of Linear DIMENSION objects (acDbRotatedDimension), control points of LEADER objects, and the infamous BLOCKDEF description property (which isn't completely available to either AutoLISP or VLISP, it is only accessible using a separately loaded function library).

Rather than bore you to tears with detailed examples, which I may do later on anyway, suffice it to say that there are still situations that warrant using AutoLISP even though Visual LISP adds so much power and potential to what you can do.

Chapter 3 – Using ActiveX with Visual LISP

In this chapter we'll discuss more examples using ActiveX capabilities within Visual LISP. First, we'll start off with the technological environment of ActiveX, including things like objects, object models, collections, properties, methods and so forth. Then we'll dig into the details of certain parts of ActiveX technologies. Understanding ActiveX functionality is essential to working with it using any language.

ActiveX is basically an object-oriented medium, meaning that it behaves in a manner that uses objects and object relationships. I am not going to explain object-oriented issues here, that's best left for more focused text books and what-not. However, I will attempt to give an overview of some of the basic object-oriented aspects for the sake of gaining a basic understanding.

Classes

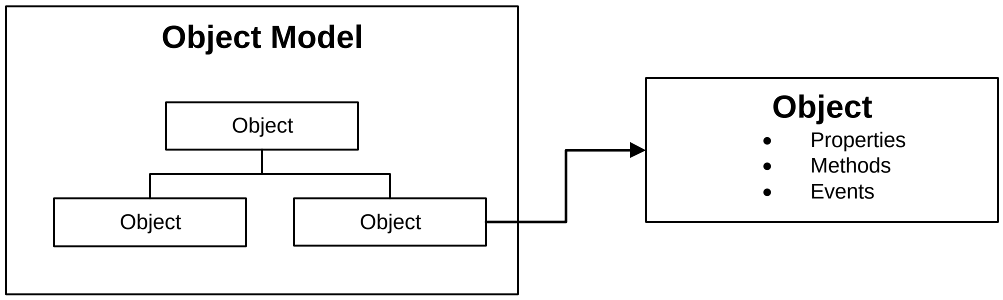
Everything in an object-oriented environment begins with *Classes*. Classes are abstract frameworks for describing what form objects should take and how they should behave and interact. Classes define categories of object types in a sense. For example, an automobile might be a class of vehicles. Vehicles could be the *parent* class and automobile would then be a *sub-class*. In turn, you can get more specific and define additional sub-classes such as station wagons, vans, and sports cars.

Classes do not address specific instances, they describe aspects about the instances in advance of their use. When you use a class you are said to *invoke an instance* of that class. The result of invoking a class is usually that of creating an object. An object can be a single entity or a container that holds yet more objects within it.

Objects

An *object* is an instance of a class. An object has inherent *properties*, and may also have inherent *methods* and *events*. Properties are attributes that define how the object behaves or reacts. Methods are built-in functions for accessing or modifying object properties or certain behaviors. Events are notifications sent by objects in response to specific actions they perform or actions that are performed upon them.

Using the automobile class example above, an object might be a particular car. Say, *your* car, which has a distinct configuration (make, model, color, options, and ID number). You could say that *your* car is an *instance* of the automobile class, or some class *derived* from the automobile class.



Object Models

An *Object Model* is an arbitrary schema, or arrangement of class relationships that define a hierarchy and means for deriving one object from a higher level of classes. An object model is independent of the languages or tools used to access it and work within its logical framework. The same model exists whether you're working with Visual Basic, VBA, Visual LISP, Delphi, Java, C/C++, C#.NET or any other language that incorporates an ActiveX interface. This does not mean that all features of the object model are equally supported in all languages. They are not. Some features are accessible or are more easily accessed within some languages than from within others.

One analogy might be that an Object Model is a house and its arrangement of rooms, doors and windows. The people that enter and use the house all deal with the same house. In this case, the house and rooms are the object model and the people are the programming languages. Hopefully, you get the point.

Class Inheritance

An Object Model always begins with a root or *base* object. In the case of AutoCAD, the base object is the AutoCAD Application object, also called the AcadApplication object. This provides the base properties, methods, events and collections from which all other objects and collections are derived. For example, the AcadApplication object has a collection of Documents (the Documents collection), which in turn has one or more Document objects within it. Each Document object has its own objects, collections, properties and methods and so on.

You can navigate an Object Model *downward* into sub-level objects and collections, as well as navigate *upward* to parent objects and collections. This *model* becomes very powerful for enabling applications to directly access and manipulate the environment to perform an almost limitless set of tasks. It also keeps things neat and organized, which always helps when developing software solutions.

Collections and Dictionaries

A *Collection* is a group of similar objects that have a common parent container. This container has a unique name and in most cases will provide its own methods for accessing the objects it contains. A *Dictionary* is a special type of collection that allows you to extend your own dictionaries. Visual LISP does not provide a means for creating or doing much with collections. It does allow you to iterate them, modify members, add and delete members. A dictionary allows you to add your own dictionaries as well as populate them, iterate them, add, modify and delete their members as well as add, modify and delete the dictionaries themselves.

Some common collections within AutoCAD are Documents, Layers, Dimension Styles, Linetypes, Blocks and so forth.

Some common dictionaries within AutoCAD are PageSetups, Layouts (yes, they are also stored as a dictionary), and various components of Express Tools such as WipeOuts. Xrecord objects are stored within dictionaries as well.

Properties, Methods and Events

Properties are simply descriptive attributes associated with an object or a collection. Examples could include Name, Height, Width, Rotation Angle, Scale, Color, Layer, Linetype and so forth. Properties will vary according to what type of object they are associated with, but some properties are common to all objects and collections. Collections and Dictionaries usually provide the Count and Name properties, as well as the Item, and Add, methods. Only dictionaries will provide a Delete method since you can't delete Collections from Visual LISP.

Methods are built-in functions that an object provides to access or modify special properties or perform special actions upon the object itself. Examples of common Methods include Rotate, Erase, Copy, Scale and Offset. You might notice that these look just like AutoCAD Modify commands. Well, in essence they are just that, but with a slight difference.

Whereas AutoCAD Modify commands are general in nature, they must validate object usage for each execution, Methods are provided by their host object and therefore, only supported Methods are provided by each object. Confused?

Stated another way, the OFFSET command can be used at any time, but if you try to OFFSET a TEXT object you'll get an error message from AutoCAD. However, the TEXT object itself provides a variety of Methods such as Copy, Rotate, Scale and Move but not Offset. So you can "invoke" a method from an object and be assured that it is valid for use with that object.

Events are actions that an object or collection can generate from various activities, which can be detected and responded to as well. This is referred to as *event-driven programming* when events are used in combination with reactions to those events. AutoCAD provides a powerful set of event-response tools called *Reactors* that enable you to post triggers in the drawing environment that respond to various Events. For example, you could create a Reactor to respond to an Erase event when an object has been erased in the active drawing. This is only one example of Events and Reactors.

Property Relevance

It is important to understand that you should NEVER assume all properties are available to all objects or collections. There are two functions in particular that are valuable for ensuring your code performs properly when dealing with properties and methods at runtime: `(vlax-property-available-p)` and `(vlax-method-applicable-p)`. These two functions are just two of the Visual LISP predicate functions that provide Boolean tests for whether a condition is True or False (*non-nil* or *nil* in LISP terminology).

The syntax for these function is as follows:

```
(vlax-property-available-p object property)
(vlax-method-applicable-p object method)
```

Properties are related to the type of object they are related to. For example, a Circle object would have a Diameter property, but Line objects would not. As an example of how Properties vary according to object types, the following code would crash with an error when picking a CIRCLE entity:

```
(if (setq ent (entsel “\nSelect object to get property: “))
  (progn
    (setq obj (vlax-ename->vla-object (car ent)))
    (princ
      (strcat “\nLength: “ (vla-get-Length obj))
    )
  )
)
```

But, if you verify that the property is valid for the relevant object first, it would perform properly as shown in the example below:

```
(if (setq ent (entsel “\nSelect object to get property: “))
  (progn
    (setq obj (vlax-ename->vla-object (car ent)))
    (if (vlax-property-available-p obj ‘Length)
      (princ
        (strcat “\nLength: “ (vla-get-Length obj))
      )
      (princ “\nObject has no LENGTH property…”)
    )
  )
)
```

Unfortunately, there is no direct means to fetch a list of all properties for a given object in such a way as to iterate it for programmatic purposes. However, you can fetch a list for informational purposes that can help you greatly.

To inquire as to what Properties and Methods a given object has you use the `(vlax-dump-object)` function on that object. The syntax of this function is `(vlax-dump-object object show-methods)` where the *show-methods* argument is either nil or non-nil. If non-nil, it shows the supported Methods for the object, otherwise Methods are simply not shown.

```
_$ (setq acadapp (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00a8a730>

_$ (vlax-dump-object (vla-get-documents acadapp) T)
; IAcadDocuments: The collection of all AutoCAD drawings open in the
current session
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;   Count (RO) = 1
; Methods supported:
;   Add (1)
;   Close ()
;   Item (1)
;   Open (2)
```

Figure 3-1 – Documents collection properties and methods.

Figure 3-1 shows the properties and methods of the Documents collection object. You'll notice that the first line of output shows the internal object reference (IAcadDocuments) along with a description of what it represents, and then it lists the available Properties and Methods.

Tip!

The following command definition may come in handy for you to explore the properties and methods of selected entities. There is no error handling provided, but it is nonetheless a useful little tool.

```
(defun C:DUMP ( / ent obj)
  (while (setq ent (entsel “\nSelect entity to get object data: “))
    (setq obj (vlax-ename->vla-object (car ent)))
    (vlax-dump-object obj T)
    (vlax-release-object obj)
  )
  (princ)
)
```

Tip!

The enclosed (RO) beside certain Properties denotes *Read-Only*, in this case all the properties are read-only. The enclosed numbers beside the Methods indicate how many arguments are required to use each method.

To access a Property, you use the `(vla-get-xxx)` function, or even the more generic `(vlax-get-Property)` function, either will work. The syntax for the first form is `(vla-get-xxx object)` where xxx is the property name. When using the `(vlax-get-property)` function ,the syntax is `(vlax-get-property object propertyname)`, where the *propertyname* can be either a double-quoted string or a single-quoted name.

```
(vlax-get-property object property) or
(vla-get-property object) or
(vlax-get object property)
```

Returns the value assigned to the named property of the object. If the property does not exist for this object, an error is generated. For example, if you request the "Diameter" property from a Line entity, this will generate an error.

Arguments:

Object – A vla-object

Property – A valid property with respect to the object.

Examples:

```
(vlax-get-property objLine "Length")
(vlax-get-property objLine 'Length)
(vla-get-Length objLine)
```

All of these expressions will do the same thing.

Property names are not case sensitive but examples throughout this book will generally capitalize the first letter for clarity. You will find that the first two options above are easiest to use in general, however, there are situations that require using the second two options. This is particularly with respect to interfacing with external applications like Microsoft Excel or Word. A fourth form `vlax-get` is a leftover from R14 for backwards compatibility only.

Using Methods

Using the example in Figure 3-1, you can see that the Documents collection object supports four Methods: *Add*, *Close*, *Item* and *Open*. The *Item* Method requires one argument to be used (hence the (1) shown beside that method in Figure 3-1), this is the index or name of the document to be fetched from the collection.

An interesting feature of the `Item` method (in general) is that it can accept either a string or an integer value argument. When given an integer argument it simply returns the (nth) item of the collection, where 0 is the first item. When given a string value, it attempts to fetch the item by its name property. The `Item(`*`name`*`)` method is not case sensitive, which is very useful for fetching names without having to first convert string cases.

Tip!

If you are familiar with Visual Basic or VBA and the use of *default methods* or *default properties*, you should be aware that this feature does not exist in Visual LISP. For example, in Visual Basic, accessing the `Item` method can be done using either of the two following ways:

```
Object.Item(12) or Object(12) or Object("Name")
```

This is because the `Item` method is the *default method* for most objects in VB or VBA. Visual LISP does not support this feature and therefore requires that you spell out all properties and methods for every use, every time. For example:

```
(vlax-invoke-method documents “Item” 12) will work…
(vla-item documents “Drawing1.dwg”) will work…
(vlax-invoke-method documents 12) will not work.
```

Using the example in Figure 3-1, the Item method might be used in any of the following ways:

- `(vla-Item documents 1)`
- `(vla-Item documents “Drawing1.dwg”)`
- `(vlax-invoke-method documents “Item” 1)`
- `(vlax-invoke-method documents ‘Item “Drawing1.dwg”)`

(vlax-invoke-method *object method [arguments]...*) or
(vla-*method object arguments*) or
(vlax-invoke *object method [arguments] ...*)

Invokes a method associated with *object* and supplies any required *arguments* to that method. If successful, returns a result. If the requested method is not provided by the object, an ActiveX error is generated. For example, requesting the “Offset” method from a Text entity, will generate an ActiveX error.

Arguments:

Object – A vla-object

Method – A method exposed by the object

Arguments – Any required arguments to supply to the method

Examples:

```
(vlax-invoke-method objLine “Move” point1 point2)
(vla-Move objLine point1 point2)
(vlax-invoke objLine “Move” point1 point2)
```

All of these of these examples do the same thing. This is generally true for most AutoCAD objects, but not for objects created from imported TypeLib interfaces or external applications or ActiveX components. You should use the first form for working with external application objects, however you can use the second form for internal objects. The third form is a leftover from R14 for compatibility only.

Tip! While you opt to use either of the two forms of Get/Put on Properties and Methods, you may find it more flexible to use the longer form (e.g. vlax-put-property) as opposed to the shorter form (e.g. vla-put-color). The reason is that by separating the property name from the function, you can define functions and iterative statements that can accept a list of properties and their associated values. For example…

```
(defun MapPropertyList (object proplist)
  (foreach propset proplist
    (if (vlax-property-available-p object (car propset))
      (vlax-put-property object (car propset) (cadr propset))
    )
  )
)
```

Be careful when trying to apply this approach to methods, as the arguments list for methods varies with respect to the object and method. Some methods don't take any arguments, while others will vary in length.

Data Types

Data types are a means for describing the type of values a given object or property can contain. Examples of data types include Integer, Double, Currency, Date, String, and so on. While AutoLISP has enjoyed *type independence* for years, Visual LISP does as well, but not always. Within AutoCAD, you can remain type independent as you can with AutoLISP, but when it comes to interacting with other applications, such as Microsoft Excel, you will inevitably have to come to terms with data types and use them wisely.

Being type independent is not a free lunch either. The price paid comes in the form of inefficient processing. When you declare data type in advance, you are telling the compiler to carve out only enough resources to suit that expected data type. To store data of type Integer for example is far less demanding than storing a "long" Date value. When you work without data types, everything is automatically allocated for the largest possible data type to make sure whatever is used will fit in the available resources. The result is that the application is more bloated than it really needs to be, both in terms of initial load size as well as runtime resource allocation.

This is essentially why applications developed in languages like C++, Java, and even Visual Basic are usually faster (when compared to similarly functional coding in type-free languages). They ensure leaner execution ahead of time in order to ensure faster performance at runtime. AutoLISP does not do this and is therefore a much slower processing language medium. Visual LISP is much better, but only if you leverage the new features to their fullest extent wherever possible.

Constants and Enumerations

A **Constant** is a special data type. It is just what it sounds like, a value that cannot be changed. This is sometimes referred to as being *static*. Often, Constants are provided by the programming language or by the hosting application itself, as a means of convenience. For example, the `acByLayer` constant can be substituted for a property value in place of 256. The name value is easier to understand and remember than an integer value. For example, the two expressions shown below are functionally identical:

```
(vla-put-color object acByLayer)
(vla-put-color object 256)
```

Enumerations are logical groups of constants that are used to identify a range of constant values. For example, you might use colors 1, 2, 3, 4, and 5 but having constants for these such as `acRed`, `acYellow`, `acGreen`, `acCyan`, `acBlue`, `acMagenta` and `acWhite` are handy for clarity as well as sensible coding. Ranges of related constant values of this type are called *enumerations*. See appendix A for a list of standard AutoCAD enumerations.

Tip!

Not all ActiveX enumerations are provided within Visual LISP. For example, the standard Decimal and Short data types are not mirrored as vlax-vbDecimal or vlax-vbShort. Refer to Chapter 6 for more information on data types.

Variants and Safearrays

In the section on Data Types above, there was mention of using the largest allocation available for type-free declarations, such as (setq) expressions in AutoLISP. Actually, this entails allocating for a ***Variant*** data type. A Variant is simply a catch-all data type that provides enough resource space to contain any other data type, be it numeric, date, string, or whatever. A Variant data type is the product of ActiveX actually, but the concept is more generic in nature and has existed long before ActiveX was around.

Visual LISP actually holds all converted ActiveX data as Variants with a specifier that denotes what specific type is contained within it. This sounds confusing but it's really very simple. The container is a Variant which is holding a Currency data type value within it. When you assign a new value to the object, you must provide that specifier to make sure that the data is properly stored. This is especially true when you are passing values between AutoCAD and other applications such as Microsoft Excel.

In addition to posting values, you can query the nested data type from a Variant value, as well as convert the value properly into a relevant LISP data type. For example, you might query a Variant object value that contains a Double value within it. You would then read that value as a REAL data type in LISP. Visual LISP provides plenty of functions for creating, reading, and modifying Variant data values in a LISP environment.

Safearrays are something like LIST objects in AutoLISP. The main difference is that they are static, meaning that they cannot be stretched or changed in terms of how many members they can store. This prevents unwanted errors generated by attempting to assign or fetch members beyond the length of the array. This is why they are called "safe" actually. Any LIST structure passed into ActiveX must be converted into a Safearray first. Any LIST oriented data fetched from an ActiveX object should be converted to a LIST data type for use by LISP functions such as (car), (nth), (assoc), (mapcar), (member) and so on. Visual LISP provides plenty of functions for creating, manipulating and reading Safearray data values.

For more information on Variants and Safearrays, refer to chapter 6.

Namespaces

A Namespace is a virtual space of resource allocation in which a process runs and interacts with other resources in that space. But it can at times communicate with other processes in other Namespaces. Think of Namespaces as bedrooms. Your application might be a person working in one bedroom; a process in a specific namespace. Another application can be working in an adjacent bedroom (namespace) as well. The two can remain independent and isolated or they can be made to pass data between each other for communication. This is essentially how namespaces work.

Some of the advantages to using namespaces are that the processes in a specific namespace are isolated from those of other namespaces and this prevents them from stepping on each other (trying to reserve resources in contention). It also enables direct loading and unloading of processes by their namespace. In other words, it's somewhat like being able to unplug one of the bedrooms from the house as if it were built in modular form. Removing one bedroom wouldn't affect the other rooms or processes they each have active.

Probably the one significant disadvantage to using namespaces is that they incur some overhead on the operating system as well as their host application. In order to manage a given namespace it has to be given its own range of memory addresses and pointer allocations. This consumes additional resources to track and control the namespace, which in turn provides the means to directly access it to unload it or pause it if necessary.

AutoCAD provides its own internal management of namespaces within Visual LISP, as well as within ObjectARX and VBA. This is yet another powerful improvement provided by Visual LISP over AutoLISP. Actually, each opened document is its own namespace as well (if you're not working in single-document mode). The effects of this can be seen when setting a variable in one drawing session and attempting to read it in another. There are ways to pass such variables between drawing sessions though, and we'll discuss these in Chapter 10.

Interfaces and Type Libraries

Interfaces are a means for connecting to the object models of other ActiveX processes or components. When you want to be able to tap into specific properties, constants, or methods of other applications, you first have to define an interface to load the object model of that target application. For example, maybe you want to be able to work with Microsoft Excel to store some AutoCAD information directly into a spreadsheet file using Excel's own tools from Visual LISP. This requires that you define the interface, and that in turn allows for the use of a *Type Library, or TypeLib.*

To use a Type Library, it must be loaded into memory and certain interface pointers have to be defined. Visual LISP provides a set of functions specifically for loading and configuring Type Library interfaces.

```
(vlax-import-type-library
     :tlb-filename name string
     :methods-prefix    string
     :properties-prefix string
     :constants-prefix  string
)
```

Imports a type library reference into the current namespace.

Arguments:

:tlb-filename string – (string) is the path and filename of the TypeLib file

:methods-prefix string – (string) is an arbitrary prefix string identifier

:properties-prefix string – (string) is an arbitrary prefix string identifier

:constants-prefix string – (string) is an arbitrary prefix string identifier

Example:

```
(vlax-import-type-library
  :tlb-filename      "c:\\myfiles\\typelibs\\tlfile.tlb"
  :methods-prefix    "dsxm-"
  :properties-prefix "dsxp-"
  :constants-prefix  "dsxc-"
)
```

This example imports the type library interface to an external application or control defined in the file tlfile.tlb. The remaining arguments define the prefixes for methods, properties and constants exposed from the type library interface.

If this type library provided a method named AddNumbers, it would be used in our Visual LISP code as *dsxm-AddNumbers*. What's interesting is that once you've actually imported the type library and this expression has succeeded, the Visual LISP will recognize all defined properties, methods and constants from the external application and color-code them in blue as with any built-in LISP function. This is another reason that the Visual LISP IDE is helpful for coding and providing features that improve your ability to spot code errors early.

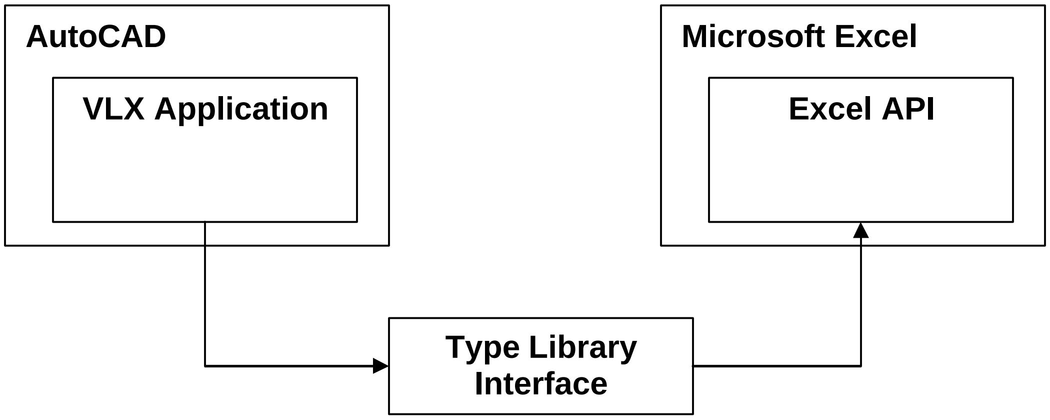


Figure 3-3 – Type Library Interfacing.

A Type Library is simply an *interface* that exposes all of the object model members of one provider to other applications that request it. When you load a type library, it immediately defines and identifies all of the publicly exposed properties, constants and methods of it's related application provider to the application consumer that is using it.

In Figure 3-3, the Excel Type Library has been loaded to interface Visual LISP with Excel's object model and use the tools it exposes. This can save a lot of time and headache by giving you direct access to tools built into Excel that will do what you need without having to attempt to reinvent the wheel in Visual LISP alone. An example of how this might be used is shown below and in Figure 3-4.

For example, when supplying a constant value as an argument to an Excel function through a call from Visual LISP, you could use the constant enumeration name instead of the actual underlying value to keep your code clear and understandable. This also saves you from having to look up all the enumerations in Excel and translating them in Visual LISP. If Excel provides a constant such as put-cellcolor you can use that directly from Excel.

Visual LISP requires TypeLib information to determine whether a method, property or constant for an object is available. Some objects may not have any TypeLib information available, such as the AcadDocument object.

`(vlax-typeinfo-available-p` *`object`*`)`

Returns T if TypeLib information is available for *object*. If none is available, it returns *nil*.

Arguments:

Object – A vla-object.

```
(defun Excel-Get-Cell (rng row column)
  (vlax-variant-value
    (msxl-get-item (msxl-get-cells rng)
      (vlax-make-variant row)
      (vlax-make-variant column)
    )
  )
)
(defun Excel-Put-CellColor (row col intcol / rng)
  (setq rng (Excel-Get-Cell (msxl-get-ActiveSheet xlapp) row col))
  (msxl-put-colorindex (msxl-get-interior rng) intcol)
)
```
Figure 3-4 – Example of using Type Library enabled code with Excel.

The second function definition in Figure 3-4 (Excel-Put-CellColor) provides a means for applying a color fill value to a given cell in an Excel worksheet from Visual LISP. This is possible by using the exposed interface methods from Excel that were provided by loading the Excel Type Library first. The type library items appear above with an **msxl-** prefix.

Once you invoke a TypeLib interface, the referenced functions are then recognized by the VLISP editor syntax engine. When you type them in properly, they change color to show that they are indeed recognized as a valid function from the external TypeLib interface. This is the basis of what makes this a useful coding practice: syntax awareness.

Tip! Type Libraries come in many forms, they are most often .TLB files, but can also be .OLB, .DLL and even .EXE files. It's worth noting that Microsoft Office 97 and 2000 normally use .TLB files, however Office XP uses the .EXE files themselves to provide the type library interface definitions to other applications. Consult the documentation for whatever external application or service you are wanting to work with for information about how it exposes its ActiveX type library information.

Chapter 4 – Debugging Code with Visual LISP

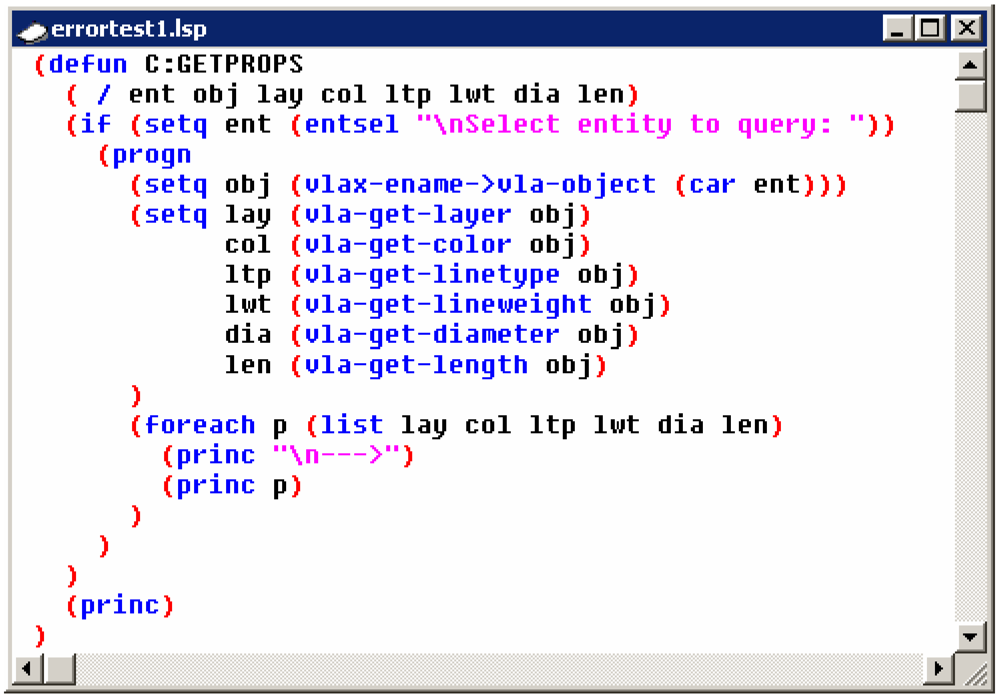
This chapter will focus on using the testing and debugging tools in Visual LISP to catch problems early and make your code more bug-proof. Debugging is as much a part of successful code development as mowing the lawn is for home ownership. It can be tedious and painful at times, but ignoring or neglecting it will certainly cause you greater pain in the long rum.

The sooner you get familiar and comfortable with using the tools for debugging, the better off you'll be in a variety of respects. Not the least of which will be improved code quality and better customer satisfaction (putting smiles on the faces of those that pay your salary never hurts).

Breakpoints

Breakpoints are a tool for placing markers in your code to trigger a pause during execution automatically. If you are having a problem with your code in a particular part of the execution, place a *Breakpoint* at the beginning of that section and run the code until it hits that Breakpoint. Then you can use one or more of the following tools to dig deeper into the execution in a methodical manner to find the cause of the problem and fix it in less time.

Example: Load the code file ERRORTEST1.LSP shown in Figure 4-1 and run the GETPROPS command in a drawing that contains a few CIRCLE, ARC and LINE entities.



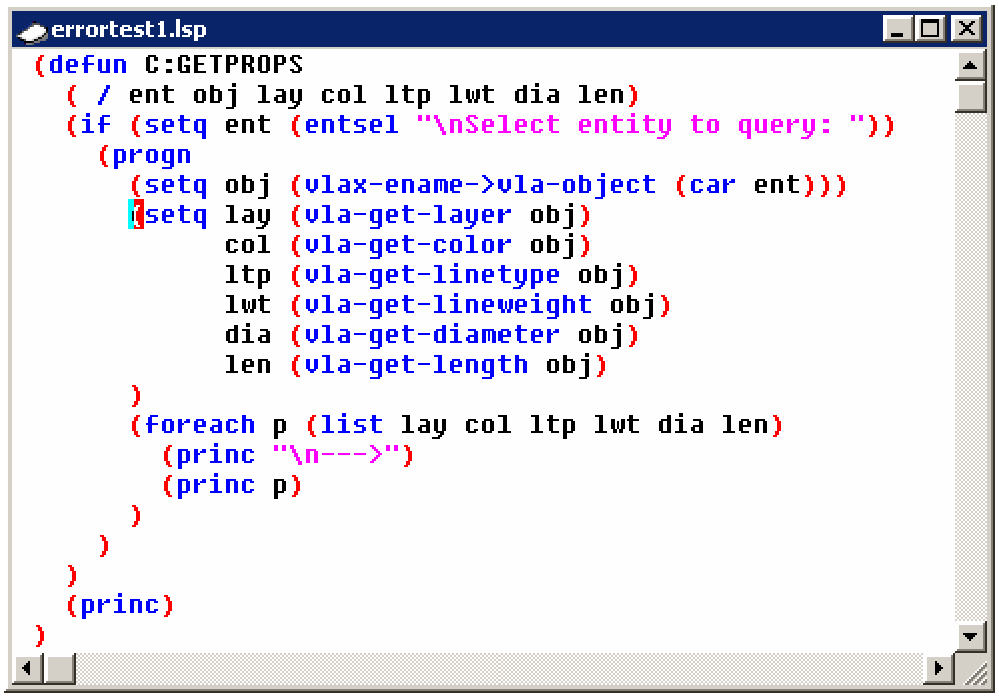
```
(defun C:GETPROPS
  ( / ent obj lay col ltp lwt dia len)
  (if (setq ent (entsel "\nSelect entity to query: "))
    (progn
      (setq obj (vlax-ename->vla-object (car ent)))
      (setq lay (vla-get-layer obj)
            col (vla-get-color obj)
            ltp (vla-get-linetype obj)
            lwt (vla-get-lineweight obj)
            dia (vla-get-diameter obj)
            len (vla-get-length obj)
      )
      (foreach p (list lay col ltp lwt dia len)
        (princ "\n--->")
        (princ p)
      )
    )
  )
  (princ)
)
```

Figure 4-1 – ERRORTEST1.LSP example code

You'll notice that when you select a Line entity, the code crashes with an error message `"error: ActiveX Server returned the error: unknown name: Diameter"`.

The same happens if you pick TEXT or POINT objects. Maybe you can see the cause of this error in the example code already, but let's pretend that this is a far more complex piece of code and you can't easily find the cause of this error by looking at the code. What to do now? Place a breakpoint in the code, load it and run it again. This time, when it gets to the breakpoint location in the code, the execution pauses, and you can begin debugging the code execution using the various tools provided in the Visual LISP IDE. One of these tools is called *Stepping*.

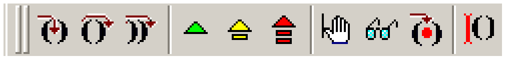
Place your editor cursor directly in front of the line that contains (setq lay …) and press the F9 key, or pick the button, to toggle the Breakpoint ON at that location. You'll see the beginning parenthesis (blocked in red. This is one of the visual aids provided by the VLISP IDE editor and it is very helpful indeed. Once you've toggle the Breakpoint ON, load the code into AutoCAD again using the button, or press CTRL+ALT+E to do the same thing.



```
(defun C:GETPROPS
  ( / ent obj lay col ltp lwt dia len)
  (if (setq ent (entsel "\nSelect entity to query: "))
    (progn
      (setq obj (vlax-ename->vla-object (car ent)))
      (setq lay (vla-get-layer obj)
            col (vla-get-color obj)
            ltp (vla-get-linetype obj)
            lwt (vla-get-lineweight obj)
            dia (vla-get-diameter obj)
            len (vla-get-length obj)
      )
      (foreach p (list lay col ltp lwt dia len)
        (princ "\n--->")
        (princ p)
      )
    )
  )
  (princ)
)
```

Figure 4-2 – ERRORTEST1.LSP with a BreakPoint set on Line #6

Now when you run the GETPROPS command and pick an entity, it will stop on that breakpoint and jump back to the VLISP IDE editor to await your next command. You'll notice here a few things are a little different now. First, the block of code that is contained within the matching parenthesis is now highlighted. Second, you'll see that the DEBUG toolbar buttons are now enabled (no longer grayed out). This toolbar is now the main tool for continuing your debugging process.



The first three buttons at left are the Stepping control buttons (described in the next section in more detail), followed by the buttons for Continue, Quit and Reset. Then the next three buttons are Toggle Breakpoint, Add Watch, and Last Break, followed last by the Breakpoint Step status button. This last button simply shows a visual queue as to whether the current process is stopped before or after a matching subset of parenthesis. It can help to look at this to know whether an error is *thrown* before the expression or just after it was evaluated.

Go ahead and pick the **Step Into** button. Keep picking on that button and watch how the code continues to execute one expression at a time. This will continue until the execution encounters the expression that generates the error. At that point, execution is aborted and the error message is displayed.

Hopefully, you'll discover that the cause of the error is that this code assumes certain properties are available without first verifying that they indeed are available. The Diameter property is obviously not available when picking a LINE entity. Nor is the Length property available when picking a CIRCLE entity.

Stepping

As you may have surmised in the above example, stepping is simply a means of walking through code execution one line or one expression at a time. This lets you pause the execution and control the frame advance to proceed along until you get to a point in the code where you want to inspect what's going on or reveal an error or specific condition. You can **Step Into**, **Step Over**, or **Step Out Of**, which are common stepping methods in all programming languages, not just Visual LISP.

Step Into – Continues advancing execution by evaluating the next expression from the innermost nested statement to the outermost statement before advancing to the next expression or statement.

Step Over – Skips the currently highlighted statement block and advances execution to the next expression or statement.

Step Out Of – Skips out of the breakpoint block and advances to the next expression or statement. If there are no more breakpoints beyond this point, execution continues uninterrupted.

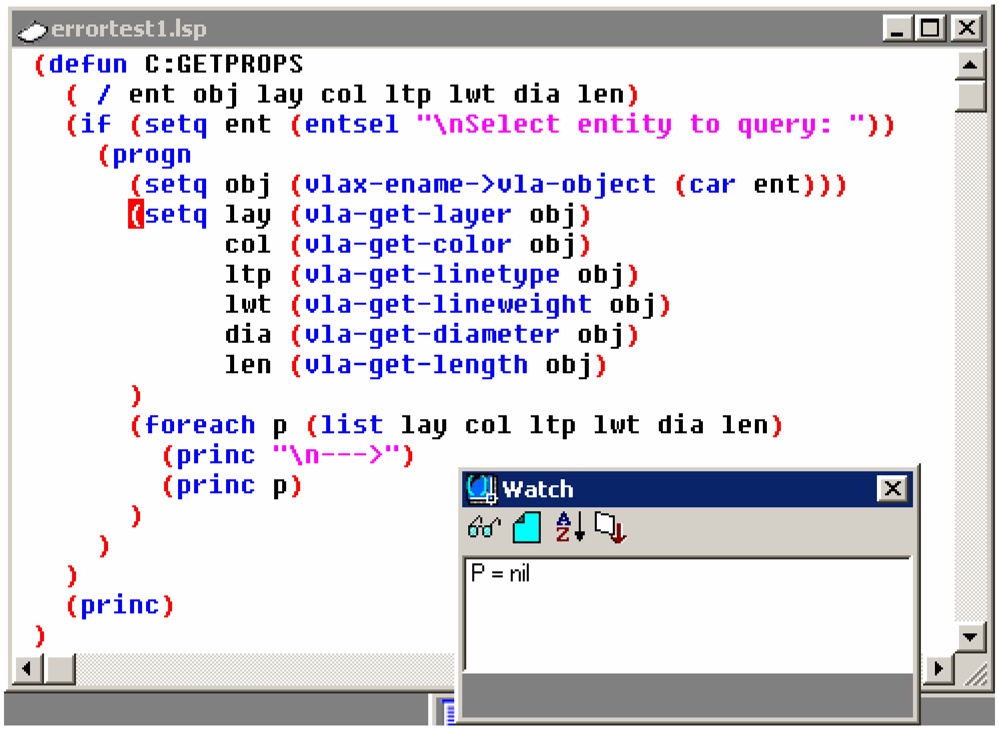
Animation

Another method of Stepping is to use Animated Execution. This feature executes the code normally but pauses after each expression is evaluated to highlight the block of code in the editor window. After each pause, the code advances automatically to the next expression. The pauses are processed using a timed delay value that you can adjust to suit your needs or preferences.

Watches

A Watch, in the context of debugging, is a marker placed on a particular object or symbol to continually display it's properties during program execution. Adding a watch to a particular variable (symbol) enables you to see its value assignment during the course of a step-execution following a breakpoint encounter in the process. To Add a Watch, select a symbol by highlighting the code in the editor window, and then pick the Add Watch button or press CTRL+W. This opens the Watch window and adds a watch reference into the watch list. You can watch as many symbols at a time as you desire, but keep in mind that the more you add, the more cumbersome it can be to clearly see what's going on.

Figure 4-3 shows a Watch being placed on the symbol 'p' in the (foreach) section of the code. By moving the breakpoint to the beginning of the (foreach) section, this will enable the Watch to display each value of the symbol 'p' as it is processed through the (foreach) iteration.



```
(defun C:GETPROPS
  ( / ent obj lay col ltp lwt dia len)
  (if (setq ent (entsel "\nSelect entity to query: "))
    (progn
      (setq obj (vlax-ename->vla-object (car ent)))
      (setq lay (vla-get-layer obj)
            col (vla-get-color obj)
            ltp (vla-get-linetype obj)
            lwt (vla-get-lineweight obj)
            dia (vla-get-diameter obj)
            len (vla-get-length obj)
      )
      (foreach p (list lay col ltp lwt dia len)
        (princ "\n--->")
        (princ p)
      )
    )
  )
  (princ)
)
```

```
P = nil
```

Figure 4-3 – Adding a Watch to the 'p' symbol.

Note that initially, P=nil, since the code is not executing and there is no value assigned to 'p' as of yet. When the (foreach) loop is entered, P will display the values for each of the symbols in the list (lay col ltp lwt dia len) respectively, even if they are set to nil.

Tracing

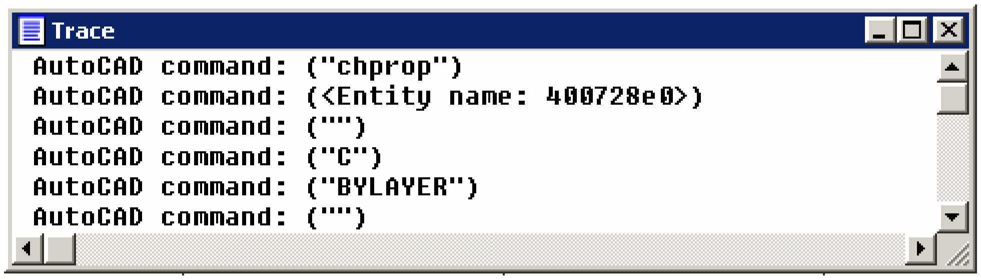
There are several Trace features provided by Visual LISP. One is a Command Trace, which places a marker on a given command (or all commands) and displays a notification in the Trace Log Window whenever the command (or any command) is called from your active code execution. If the VLISP IDE is open, the Trace log window is displayed and any calls are posted there during execution.

If the VLISP IDE is not active, the trace dump is posted to the AutoCAD command prompt window. However, once Visual LISP (IDE) is activated, it remains active even when you return to the AutoCAD editor session. Therefore, once VLISP is activated, all Trace output is sent to the Trace window in the VLISP IDE and you must return to the VLISP IDE session to continue viewing trace output until you close and reopen AutoCAD to terminate the trace output to VLISP.

Another type of Trace is a Stack Trace.

The Trace button is not on the DEBUG toolbar, but instead on the VIEW toolbar. This is because the Trace feature is actually a window display as opposed to a debugging command related to a specific piece of code (as is the Add Watch and Breakpoint features).

To display the Trace Log window, you must first turn ON the command trace by picking the Debug pulldown menu and checking the option titled “Trace Command”. Once this is done, any calls to AutoCAD commands from your code execution in the VLISP IDE are reported to the Trace Log window as shown below.



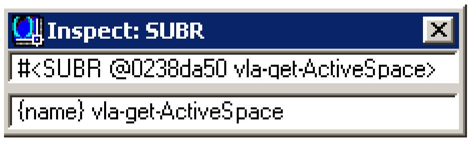
```
AutoCAD command: ("chprop")
AutoCAD command: (<Entity name: 400728e0>)
AutoCAD command: ("")
AutoCAD command: ("C")
AutoCAD command: ("BYLAYER")
AutoCAD command: ("")
```

Figure 4-4 – Trace Log window after CHPROP command called.

Figure 4-4 shows how a command such as CHPROP is reported to the Trace Log window along with any arguments it uses such as entity name, command-line options and values supplied to it. You may notice that each component is represented as a single-member list. This is because VLISP represents command stacks in list form internally.

Inspection

Inspection involves drilling down into a symbol to see what properties it contains and what form it is defined as. For example, inspecting the function (vla-get-activespace) will show that it is defined as a SUBR, which is an intrinsic or built-in function provided by Visual LISP. The number/letter string to the right of the declaration denotes it's memory address in the current namespace.

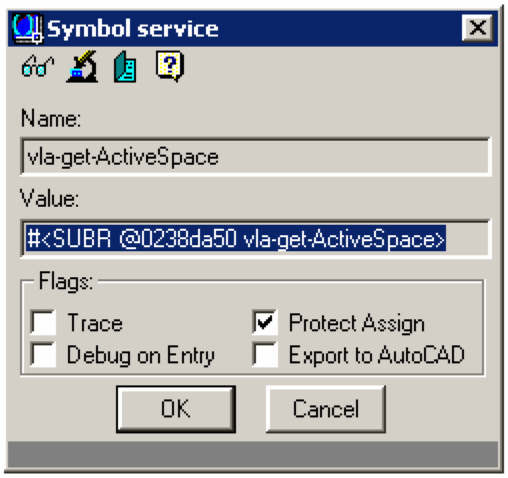

Inspect: SUBR

#<SUBR @0238da50 vla-get-ActiveSpace>

{name} vla-get-ActiveSpace

Symbol Service

The Symbol Service utility provides a way to inspect symbols as to their properties. This includes protection status, tracing, debug on entry status and whether it has been exported to the AutoCAD namespace. From this pop-up form you can also perform online help look-ups by picking the help button at the top of the form. The example below shows the result of highlighting the code (vla-get-activespace) and picking the Symbol Service button. You can also right-click on the highlighted code and pick Symbol Service from the pop-up menu.


Symbol service

Name:

vla-get-ActiveSpace

Value:

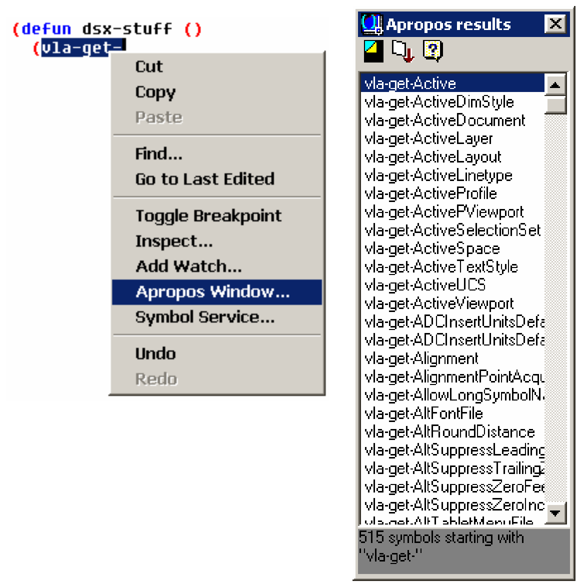
#<SUBR @0238da50 vla-get-ActiveSpace>

Flags:

- [ ] Trace
- [✓] Protect Assign
- [ ] Debug on Entry
- [ ] Export to AutoCAD

Apropos

The APROPOS feature allows you to search for functions, properties and methods based upon wildcard matches and returns a list of them within a listbox in the IDE. From this list you can copy/paste into your code window or perform online help lookups to learn what the item can do or how it's used. There are various ways to invoke this feature. One of them is to right-click on some code and pick Apropos from the pop-up menu (shown in the example below). Or you can pick the Apropos button on the View toolbar.

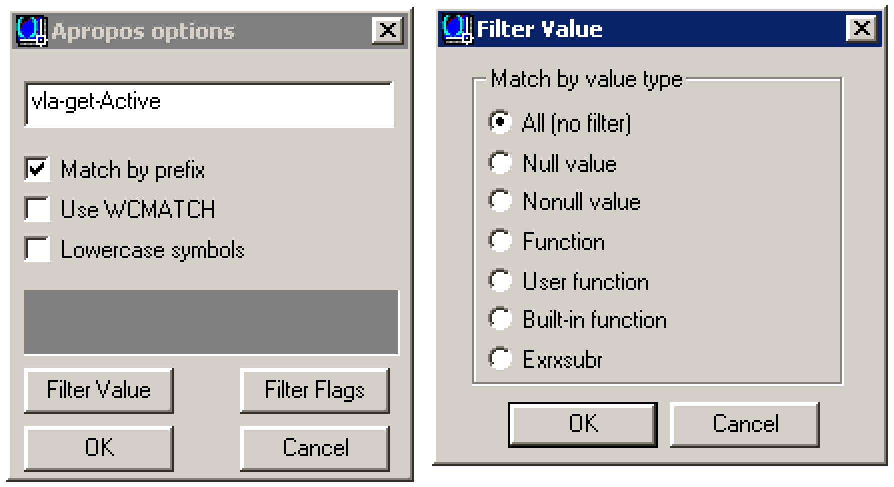


```
(defun dsx-stuff ()
  (vla-get-
```

```
vla-get-Active
vla-get-ActiveDimStyle
vla-get-ActiveDocument
vla-get-ActiveLayer
vla-get-ActiveLayout
vla-get-ActiveLinetype
vla-get-ActiveProfile
vla-get-ActivePViewport
vla-get-ActiveSelectionSet
vla-get-ActiveSpace
vla-get-ActiveTextStyle
vla-get-ActiveUCS
vla-get-ActiveViewport
vla-get-ADCInsertUnitsDefa
vla-get-ADCInsertUnitsDefa
vla-get-Alignment
vla-get-AlignmentPointAcqu
vla-get-AllowLongSymbolN
vla-get-AltFontFile
vla-get-AltRoundDistance
vla-get-AltSuppressLeading
vla-get-AltSuppressTrailing
vla-get-AltSuppressZeroFee
vla-get-AltSuppressZeroInc
vla-get-AltTabletMenuFile
515 symbols starting with
"vla-get-"
```

As you can see by the example above, an APROPOS search on “vl-get-“ turns up quite a few matching items in the Results window. You can narrow down the search by typing in a few more characters in your code window, such as “vla-get-Active” to only find those items that begin with the same string value.

You can also modify the Apropos search within the Results window by picking on the top-left button (tool tip says “Apropos Options”) and entering your changes to the search criteria in the edit box. Other options on this form allow you to specify case-matching, prefix only, and lowercase conversion. The Filter Value button displays even more options on the Filter Value form.



Apropos options

vla-get-Active

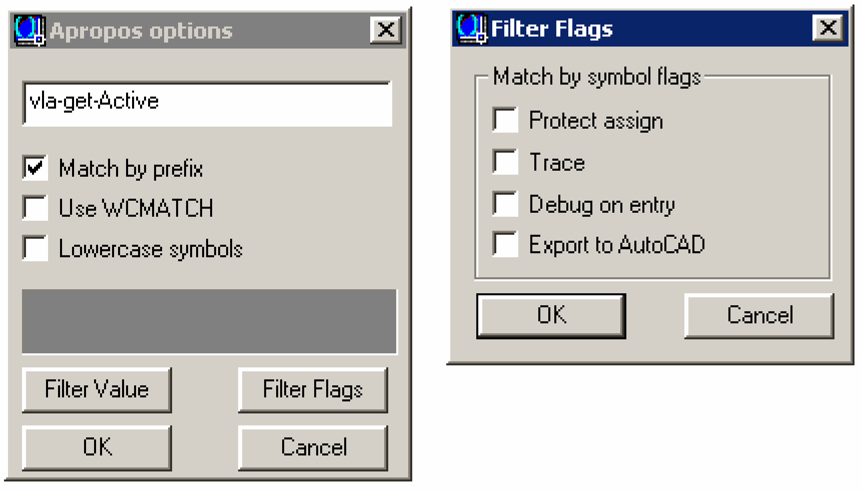
[✓] Match by prefix
[ ] Use WCMATCH
[ ] Lowercase symbols

Filter Value

Match by value type
[✓ All (no filter)]
Null value
Nonull value
Function
User function
Built-in function
Exrxsubr

Figure 4-5 – Apropos Filter Value options

You can, for example, limit your search to items such as built-in functions, externally-defined functions (ExrSubrs such as those defined by ObjectARX applications), and Null or Non-Null values. The Filter Flags button displays a search filtering form for limiting the search to symbols that have certain characteristics themselves, such as being protected or those that have been exported to the AutoCAD namespace.



vla-get-Active

- [✓] Match by prefix
- [ ] Use WCMATCH
- [ ] Lowercase symbols

- [ ] Protect assign
- [ ] Trace
- [ ] Debug on entry
- [ ] Export to AutoCAD

Figure 4-6 – Apropos Filter Flags options

Bookmarks

Bookmarks are not necessarily a *debugging* tool, but they are useful for locating a particular section of code quickly. This is especially true in cases where you are working with very large amounts of code in a single file and it becomes difficult to jump around in the file to specific points in the code. Bookmarks appear as a rounded square solid green symbol in front of the line where you insert them.

To insert a Bookmark, place the cursor on the desired line of code and press ALT+. (a period) or pick Search/Bookmarks/Toggle Bookmark. To remove a bookmark, place the cursor on the bookmarked line and press ALT+. again or pick Search/Bookmarks/Toggle Bookmark. To clear all bookmarks in a given file, pick Search/Bookmarks/Clear All Bookmarks.

Tip! While Visual LISP does not allow you to jump to bookmarks by name, you can move between them in a Next/Previous manner. To jump from one bookmark to the next, press CTRL+. (a period). To move to the previous bookmark, press CTRL+, (a comma) or continue pressing CTRL+. Until you cycle through all the bookmarks again.

Goto Line Position

When Bookmarks are not practical, you can also jump directly to a line in your code by number. Simply press CTRL+G to display the Goto Line box, enter the line number and press Enter to go to that line.

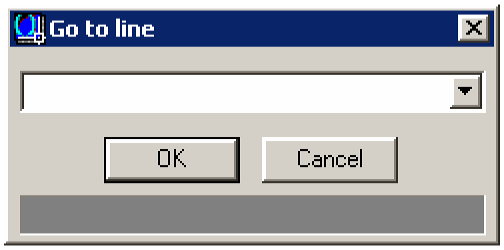


Figure 4-7 – The Go To Line Box.

Error Trapping

Ultimately, no method of debugging will get you to the goal line without proper *error trapping*. What is Error Trapping? It is simply a process of capturing an error in order to diagnose the nature of the error and performing some corrective action as a result. This is more efficient and produces better quality results than simply allowing the error to crash your code and display an ugly, cryptic message that confuses the user.

ActiveX in particular, is not known for being very friendly when it comes to the content of its error messages. For example, a common error message thrown by ActiveX operations in Visual LISP is the following:

```
Error: ActiveX error: No description provided.
```

What does this mean to the user? For that matter, what does it mean to anyone? Not much. However, within the context of your code, you might be trying to initiate a connection to an Access database using ADO or JET. At the point where you would try to make the connection, you should place an Error Trap around that code and test whether it succeeded or failed, and if it failed, determine why it failed. Then you can check the error conditions and display a meaningful message that may help the user figure out the cause themselves, saving you even more work.

How do you place an Error Trap around your code? You use the functions provided by Visual LISP for trapping, checking and handling errors generated by an ActiveX object.

Visual LISP Error Trapping Functions

Visual LISP provides some additional error trapping and error handling functions over the age-old AutoLISP *error* function. Each of these functions give you a collective set of tools to catch, verify and handle errors thrown from code execution in Visual LISP, especially for code that runs in it's own namespace or that is interfacing with external application objects or procedures. For example, it can be very difficult at times to intercept and react to errors generated from things like ADO failures unless you use these special functions.

(vl-catch-all-apply '*function list*)

Places an error trap over the result of a function execution. Works similarly to the Try-Catch exception handing provided in C++, C# and VB.NET programming languages. Returns either the successful object or an Error object. The `(vl-catch-all-error-p)` function determines if the return object is an Error object.

Arguments:

Function – Either a defun or lambda function definition or symbol pointer

List – A list of required arguments for the function being evaluated

The `(vl-catch-all-apply)` function is used to place an error "catch" (trap) around a set of code expressions. Once executed, any result is passed directly to the output of this function where it can be checked to see if it generated an error, and if so, what kind of error was generated.

The syntax for this function is `(vl-catch-all-apply` ***`function list`***`)` where *function* is the expression being executed, and *list* is the items on which the function is being executed upon or by way of.

Tip!

Be aware of every ActiveX object you intend to use or interface with. You should be careful to determine whether or not the object will "throw" an ActiveX or OLE error when it fails. If it is capable of throwing such an error (as opposed to returning nil) as the result of a failure, you should ALWAYS wrap the expressions used to interface with it inside of a error handler to keep your code from "blowing up" on the user.

For example, to place an error trap around an attempt to open Microsoft Excel, you could use something like this…

```
(cond
  ( (vl-catch-all-error-p
      (setq XL
        (vl-catch-all-apply
          'vlax-create-object
         '("Excel.Application")
       )
      )
    )
    (vl-exit-with-error
      (strcat "\nError: " (vl-catch-all-error-message XL))
    )
  )
  ( T (princ "\nSuccessfully opened Microsoft Excel session object.") )
)
```

Figure 4-8 – Error trapping example using an Excel application object

This small example does the following (working from the inside out in order of processing):

- Attempt to create an object of Excel.Application

- If the attempt fails it returns appsession as an Error object.
- `(vl-catch-all-error-p)` returns T when it inspects appsession
- Evaluation is aborted by the `(vl-exit-with-error)` function which displays the error message passed through the Error object appsession.
- This error causes the code to abort execution immediately and displays the message to the user at the same time. Otherwise, if `(vl-catch-all-error-p)` returns nil, the returned appsession object is not an Error object, and the program can continue on to do more things with it.

A more simple and direct test is to force a "Divide by Zero" failure to create an error and see how Visual LISP handles it. From within the LISP Console window, enter the following two lines of code in the order shown. After the first line, you should see the error object returned as **`<%catch-all-apply-error%>.`** After the second line, you should see the string value message returned from the Error object as "divide by zero".

```
_$ (setq catchit (vl-catch-all-apply '/ '(50 0)))
#<%catch-all-apply-error%>
_$ (vl-catch-all-error-message catchit)
"divide by zero"
```

A good place to use `(vl-catch-all-apply)` is when attempting to fetch an object from a collection using the `(vla-item)` method. For example, you might expect the following code fragment to return nil if no matching object is found. However, this fragment would throw an ActiveX error instead.

```
(setq layers
  (vla-get-layers
    (vla-get-activedocument (vlax-get-acad-object))))
(setq mylayer (vla-item layers "Doors"))
```

The proper way to do this would be to use `(vl-catch-all-apply)` to trap the error when a request fails. This might look something like the following:

```
(if
  (not
    (vl-catch-all-error-p
      (setq mylayer
        (vl-catch-all-apply 'vla-item (list layers "Doors"))
      )
    )
  )
  (princ "\nLayer was found in layers collection!")
  (princ "\nLayer does not exist.")
)
```

Tip!

Here is an example function that I will use throughout this book in place of `(vla-item)`. It returns an object or nil if no item is found in the provided collection. I highly recommend using a function like this in place of `(vla-item)` to avoid errors in your code.

```
(defun get-item (collection item / result)
  (if
    (not
      (vl-catch-all-error-p
        (setq result
          (vl-catch-all-apply 'vla-item (list collection item))
        )
      )
    )
    result
  )
)
```

(vl-catch-all-error-p *object*)

Returns T or nil depending upon whether *object* is an Error object or not.

Arguments:

Object – Any vla-object

Example:

```
(vl-catch-all-error-p (vl-catch-all-apply '/ '(50 0)))
```

This will returns `T` (true) because (/ 50 0) is a classic “divide by zero” failure.

(vl-catch-all-error-message *object*)

Returns the message description from an Error object. If *object* is not an Error object, this function returns *nil*.

Arguments:

Object – Any vla-object

Example:

```
(vl-catch-all-error-message (vl-catch-all-apply '/ '(50 0)))
```

This will display an error message “divide by zero”.

(vl-exit-with-error *message*)

Aborts the VLX execution and returns a string message result.

Arguments:

Message – A string containing an error message result

The `(vl-exit-with-error)` function aborts execution immediately and returns a string value as the result. This is useful for passing up a custom error message that may provide added clarity to users. This works very much like the AutoLISP `(exit)` function except that you can pass a return value back as a result of the error. Figure 4-8 shows how you can pass `(vl-catch-all-error-message)` as the return message value.

(vl-exit-with-value *value*)

Aborts the VLX execution and returns a numeric or symbolic result value.

Arguments:

Value – Any value or symbol

Example:

```
(defun fubar (somevalue / *error*)
  (defun *error* (s)
    (vl-exit-with-value s)
  )
  (/ somevalue 0); force divide by zero error
)

(defun errortest ( / try)
  (cond
    ( (vl-catch-all-error-p
        (setq try (vl-catch-all-apply 'fubar (list 12)))
      )
      (princ (strcat "\nError: " (vl-catch-all-error-message try)))
    )
  )
)
```

If you load the above example and type in (errortest), the result will be "Error: divide by zero". The `(vl-exit-with-value)` function works the same way as `(vl-exit-with-error)` except that it returns a numeric value as the result. This can be helpful if you want to handle errors using a numeric value parameter, such as passing up the return value of an ActiveX Error number.

As you can see from these functions and the figures shown, you can perform very detailed error trapping and handling using Visual LISP to help you produce better quality code and software products. This practice is not unique to Visual LISP by any means. It is the same in general as what is done with other languages such as C/C++, Visual Basic, VBA, Java and so forth. Error trapping makes sense but you have to make the effort to put it to efficient use to get the benefits it offers.

Chapter 5 – Working with Projects and Multiple Files

Projects are collections of related LSP files that you associate together for a common purpose. Examples can be multiple files that comprise a single feature or group of features that you want to always be working together in some respect. This is also called a *Work Space* in other products, but the overall intent is the same: Collect related code files together with a name to make it easier to open and work on them together.

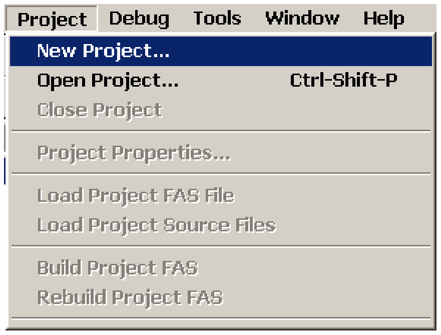
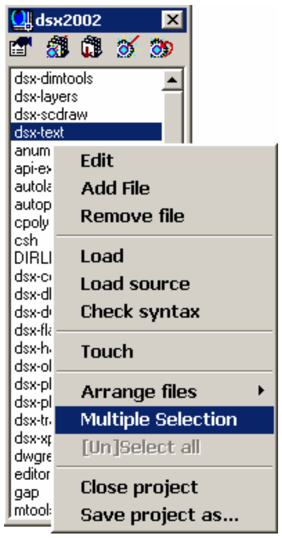


Figure 5-1 – The Project pulldown menu

While Projects are great, VLISP has certain limitations that make them less than ideal compared to other code development tools on the market such as Microsoft Visual Studio. Among these limitations are that you cannot include DCL or other types of code files, and you can only compile the project to FAS output, not to VLX output. Ideally, a VLISP project should allow for all file types that can be included in a VLX application (DVB, txt, LSP, DCL).



Nonetheless, Projects are very useful for nothing less than to keep related LSP files together and be able to quickly open any or all of them in the editor. See Figures 5-2 and 5-3 for an example of how a project configuration is managed.

Once you create and open a project, it will display a dockable listbox in the VLIDE window that displays all the member .LSP files. To open a particular file, simply double-click on it.

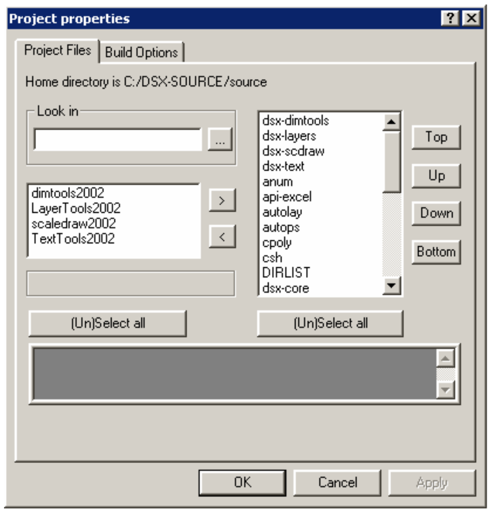
You can also open multiple files at once by right-clicking on the list and picking Multiple Selection from the pop-up menu (refer to the image at right).

You can add and remove files from a project at any time by using the Add File or Remove File options from the pop-up menu. You can also add or remove files from the Project Properties form (see Figure 5-2).

Tip!

The order in which you add files or sort them in the project files list is the order they will be compiled in when using a project files list as the input for the Make

Application Wizard (discussed in Chapter 13). You can go back and modify the order of files after they have been added into a given project.


Project properties

Project Files | Build Options

Home directory is C:/DSX-SOURCE/source

Look in

dimtools2002
LayerTools2002
scaledraw2002
TextTools2002

dsx-dimtools
dsx-layers
dsx-scdraw
dsx-text
anum
api-excel
autolay
autops
cpoly
csh
DIRLIST
dsx-core

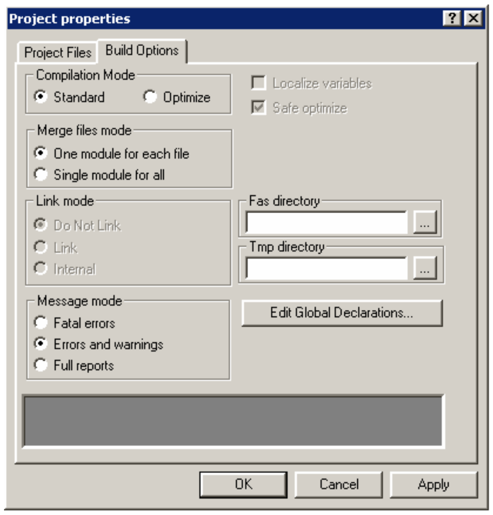
Top | Up | Down | Bottom

(Un)Select all | (Un)Select all

OK | Cancel | Apply

Figure 5-2 – The Project Properties form, Files tab

Figure 5-2 shows the main properties form for making and modifying a Visual LISP project. Note that there are two tabs "Project Files" and "Build Options". Figure 5-3 shows the "Build Options" tab panel. The Project Files panel is where you select the .LSP files to be part of your project.


Project properties

Project Files | Build Options

Compilation Mode: [✓ Standard] Optimize

Localize variables

[✓ Safe optimize]

Merge files mode: [✓ One module for each file] Single module for all

Link mode: [✓ Do Not Link] Link Internal

Fas directory

Tmp directory

Message mode: Fatal errors [✓ Errors and warnings] Full reports

Edit Global Declarations...

OK | Cancel | Apply

Figure 5-3 – Project Properties / Build Options

The options shown in Figure 5-3 will be explained in more detail in Chapter 13 ("Making Applications"). All of these options pertain to the making of FAS output files, .FAS files are compiled LISP code that can be created from one or more LSP files as a single .FAS file, which can be compiled into a VLX application file with other FAS files.

Chapter 6 – Working with Variants and Safearrays

While the topic of Variant and Safearray data types was discussed earlier in this book, they are significant enough within the ActiveX world of Visual LISP to warrant an entire chapter devoted solely to them. We will begin by briefly reviewing what they are and then begin exploring how to work with them using Visual LISP functions.

As we mentioned before, a Variant is a data type that is designed to be a generic catch-all container for any other type of data. They consume the most memory and processing resources of all the data types because they are the largest in terms of resource requirements.

Languages such as C/C++, Visual Basic and Delphi provide declaration statements to notify the compiler in advance as to what data types each variable will contain. This not only guarantees leaner resource requirements but also allows for error checking during compilation that heads off runtime problems.

Visual LISP Variant Functions

`(vlax-make-variant [value] [type])`

Creates a variant object using the given value or symbol evaluation

Arguments:

Value – The value to be assigned to the variant. If value is omitted, an empty variant of type vlax-vbEmpty is created.

Type – The data type of the variant. If type is omitted, the LISP data type is cast to the closest ActiveX data type (see table below).

Examples:

```
(vlax-make-variant) or (vlax-make-variant nil)
```
Creates an *uninitialized* variant of type `(vlax-vbEmpty).`

```
(vlax-make-variant 10 :vlax-vbInteger)
```
Creates a variant of type Integer (`vlax-vbInteger`) with value of 10.

```
(vlax-make-variant “vlisp example”)
```
Creates a variant of type String (`vlax-vbString`) with value of “vlisp example”.

```
(setq dblarray (vlax-make-safearray vlax-vbDouble ‘(0 . 3)))
(vlax-make-variant dblarray :vlax-vbArray)
```
Creates a variant containing a safearray of double values.

Tip! The Decimal, and Short ActiveX data types are not supported in Visual LISP. You can however, specify their types using the (vlax-variant-type) when reading in values from external sources. To send data to external sources in these types, you may have to use the numeric representation of (vlax-vbDecimal) and (vlax-vbShort) as they are not provided as enumerations within Visual LISP. For example, a Decimal data type is enumeration value 14.

Variant Data Types

What if you don't specify the data type for the Variant constructor? Visual LISP will attempt to convert it to an appropriate variant data type using a default mapping. Table 6-1 below shows the default mapping of data types from LISP to Variant.

LISP Data Type	Variant Default Data Type Assignment
nil	vlax-vbEmpty
:vlax-null	vlax-vbNull
INT (integer)	vlax-vbLong
REAL (float)	vlax-vbDouble
STR (string)	vlax-vbString
VLA-OBJECT	vlax-vbObject
:vlax-true or :vlax-false	vlax-vbBoolean
VARIANT	Same as the type of initial value
SafeArray	vlax-vbArray
N/A	vlax-vbShort
N/A	vlax-vbDecimal
N/A	vlax-vbDate

Table 6-1 – Visual LISP Default LISP->Variant Data Mappings

(vlax-variant-type variant)

Returns the data type of a variant. If the *symbol* is not a variant, an error is generated. The return value is an enumeration of the data type (see Appendix A for Data Type enumerations).

Arguments:

Symbol – A symbol containing a variant value.

Examples:

```
(setq vartest (vlax-make-variant 6 vlax-vbInteger))
(vlax-variant-type vartest) returns 2 (integer type)
(setq vartest (vlax-make-variant "dog" vlax-vbString))
(vlax-variant-type vartest) returns 8 (string type)
```

(vlax-variant-value *symbol*)

Returns the value contained in a variant symbol. If *symbol* does not contain a variant data type, and error is generated. Otherwise, the data type is returned as an enumeration (integer) value (see Appendix A for Data Type enumerations).

Arguments:

Symbol – A symbol containing a variant value.

Examples:

```
(setq vartest (vlax-make-variant "testvalue" vlax-vbString))
(vlax-variant-value vartest)
```
Returns value "testvalue" as a string result.

```
(setq sa (vlax-make-safearray vlax-vbDouble '(0 . 2)))
(setq vartest (vlax-make-variant sa vlax-vbDouble))
(vlax-variant-value vartest)
```
Returns a value of #<safearray…> which is a vla-object type.

```
(vlax-safearray->list (vlax-variant-value vartest))
```
Returns value (0.0 0.0 0.0) a list result.

(vlax-variant-change-type *symbol type*)

Changes the data type assignment of a variant.

Arguments:

Symbol – A variant value

Type – Data type number or enumeration to convert to

Examples:

```
(setq vartest (vlax-make-variant 5 vlax-vbInteger))
(setq vartest (vlax-variant-change-type vartest vlax-vbString))
```

Converts vartest to variant of type String (vlax-vbString) which would result in a value being returned from (vlax-variant-value) as "5".

Visual LISP SafeArray Functions

(vlax-make-safearray *type dim1 [dim2] …*)

Creates a safearray of data type *type* of dimension bounds *dim1*, etc. where additional dimensions can be specified. If the operation fails for any reason, expression returns *nil*.

Arguments:

Type – Data type (integer or enumeration)

Dim1 – Dimension of array (one dimension array)

Dim2 – (optional) Dimension of second array (two dimension array) etc.

Examples:

```
(setq sa (vlax-make-safearray vlax-vbDouble '(0 . 2)))
```

Creates a single-dimension array of doubles, capable of storing three distinct elements (0, 1, 2).

```
(setq sa (vlax-make-safearray vlax-vbString '(0 . 1) '(1 . 3)))
```

Creates a two-dimensional array of strings, the first dimension contains two elements beginning at index 0. The second dimension contains three elements and begins at index 1.

Tip! To populate a SafeArray you can either use `(vlax-safearray-fill)` or `(vlax-safearray-put-element)` depending upon whether you need to assign elements one at a time or all at once.

(vlax-safearray->list symbol)

If *symbol* contains a safearray, the elements are returned in a LISP LIST data type. If symbol if does not contain a safearray, an error is generated. You should wrap calls to this function inside of an error catch to ensure proper error handling.

Arguments:

Symbol – A symbol containing a safearray

(vlax-safearray-type *symbol*)

If *symbol* contains a safearray, the data type of the elements is returned as an enumerated result (integer value). This can be matched either by the integer or enumeration result (see Appendix X for Data Type enumerations). If *symbol* does not contain a safearray, an error is generated.

Arguments:

Symbol – A symbol containing a safearray

Examples:

```
(setq sa (vlax-make-safearray vlax-vbdouble '(0 . 3)))
(vlax-safearray-type sa)
```

Returns 5 (a double) which equates to vlax-vbDouble

`(vlax-safearray-fill safearray 'element-values)`

Assigns values to multiple elements in a safearray. If the supplied argument is not an array, an ActiveX error is returned. You should wrap calls to this function inside of an error catch to ensure proper error handling.

Arguments:

Safearray	An object of type safearray.
Element-values	A list of values to be stored in the array. You can specify as many values as there are elements in the array. If you specify fewer values than there are elements, the remaining elements retain their current value or are left empty. For multi-dimension arrays, *element-values* must be a list of lists, with each list corresponding to a dimension of the array.

Create a single-dimension array of *double* values:

```
_$ (setq myarray (vlax-make-safearray vlax-vbdouble '(0 . 2)))
#<safearray...>
```

Use vlax-safearray-fill to populate the array elements with values:

```
_$ (vlax-safearray-fill myarray '(1 2 3))
#<safearray...>
```

List the contents of the array to verify element values:

```
_$ (vlax-safearray->list myarray)
(1.0 2.0 3.0)
```

`(vlax-safearray-get-element safearray element [element...])`

Returns the value of specified elements within a safearray, where element values are integers denoting the index locations to fetch within the array. If the safearray argument is not a safearray object, an ActiveX error is returned. You should wrap calls to this function inside of an error catch to ensure proper error handling.

Arguments:

Safearray	An object of type Safearray
Element	Integer of index location to fetch

```
_$ (setq sa (vlax-make-safearray vlax-vbString '(1 . 2) '(1 . 2) ))
```

```
#<safearray...>
```

Use vlax-safearray-put-element to populate the array:

```
_$ (vlax-safearray-put-element sa 1 1 "A")
"a"
_$ (vlax-safearray-put-element sa 1 2 "B")
"b"
_$ (vlax-safearray-put-element sa 2 1 "C")
"c"
_$ (vlax-safearray-put-element sa 2 2 "D")
"d"
```

Use vlax-safearray-get-element to retrieve the second element in the first dimension of the array:

```
_$ (vlax-safearray-get-element sa 1 1)
"A"
_$ (vlax-safearray-get-element a 2 2)
"D"
```

(vlax-safearray-put-element *safearray element [element...] value*)

Assigns a new value to a single element in a safearray. If the safearray argument is not a Safearray object, an ActiveX error is returned. If the element-value supplied is not capable of casting into the expected array data type, an ActiveX error is returned. You should wrap calls to this function inside of an error catch to ensure proper error handling.

Arguments:

Safearray	An object of type Safearray
Element	A set of index values pointing to the element you are assigning a value to. For a single-dimension array, specify one index value; for a two-dimension array, specify two index values, and so on.
Value	A value to assign to each element. To assign different values to individual elements in the array, make separate calls with unique values to correspond to the appropriate element locations.

```
_$ (setq sa (vlax-make-safearray vlax-vbString '(1 . 2) '(1 . 2) ))
#<safearray...>
```

Use vlax-safearray-put-element to populate the array:

```
_$ (vlax-safearray-put-element sa 1 1 "A")
"A"
_$ (vlax-safearray-put-element sa 1 2 "B")
```

```
"B"
_$ (vlax-safearray-put-element sa 2 1 "C")
"C"
_$ (vlax-safearray-put-element sa 2 2 "D")
"D"
```

You can also populate array values using the `vlax-safearray-fill` function. The following function call accomplishes the same task as three `vlax-safearray-put-element` calls:

```
(vlax-safearray-fill sa '(("A" "B") ("C" "D")))
```

(vlax-safearray-get-dim *safearray*)

Returns the dimension (number of array dimensions) in a given safearray. If the supplied argument is not an array, an ActiveX error is returned. You should wrap calls to this function inside of an error catch to ensure proper error handling.

Arguments:

Safearray	An object of type Safearray

```
_$ (setq myarray (vlax-make-safearray vlax-vbinteger '(2 . 5)))
#<safearray...>

_$ (vlax-safearray-get-dim myarray)
2
```

(vlax-safearray-get-l-bound *safearray dim*)

Returns the lower boundary of the specified array dimension (an integer value). If the supplied argument is not an array, an ActiveX error is returned. You should wrap calls to this function inside of an error catch to ensure proper error handling.

Arguments:

Safearray	An object of type Safearray
Dim	The integer location of the dimension within the array, where the first dimension is 1.

The following examples evaluate a safearray defined as follows:

```
(vlax-make-safearray vlax-vbString '(1 . 2) '(0 . 1) ))
```

Get the starting index value of the array's first dimension:

```
_$ (vlax-safearray-get-l-bound tmatrix 1)
1
```

(vlax-safearray-get-u-bound *safearray dim*)

Returns the upper boundary of the specified array dimension (an integer value). If the supplied argument is not an array, an ActiveX error is returned. You should wrap calls to this function inside of an error catch to ensure proper error handling.

Arguments:

Safearray	An object of type Safearray
Dim	The integer location of the dimension within the array, where the first dimension is 1.

```
(setq sa (vlax-make-safearray vlax-vbString '(1 . 2) '(0 . 1) ))
_$ (vlax-safearray-get-u-bound sa 1)
2
```

The first dimension ends with index 2.

Get the end index value of the second dimension of the array, which starts at 1:

```
_$ (vlax-safearray-get-u-bound sa 2)
1
```

Chapter 7 –Object Manipulation Functions

Visual LISP provides a set of functions that allow you to create, manipulate and close ActiveX objects. This is normally with respect to external application session objects, but it can also apply to any external process object, such as DLL or OCX interfaces.

(vlax-get-object *program-id*)

Attempts to connect to an existing object (process). Same as the Visual Basic/VBA function GetObject (*program-id*).

Arguments:

Program-ID – A string that names the application object class identifier. For example “Word.Application” or “Excel.Application”.

Example:

```
(setq xlapp (vlax-get-object “Excel.Application”))
```

Returns a vla-object to the external Excel application session if successful, otherwise it returns *nil*.

(vlax-create-object *program-id*)

Attempts to create a new object session (process). Same as the Visual Basic function CreateObject (*program-id*).

Arguments:

Program-ID – A string that names the application object class identifier. For example “Word.Application” or “Excel.Application”.

Example:

```
(setq xlapp (vlax-create-object “Excel.Application”))
```

Returns a vla-object to the new external Excel application session if successful, otherwise it returns *nil*.

(vlax-get-or-create-object *program-id*)

Attempts to first connect to an existing object session, and then, if none are found, it attempts to create a new object session. This function has no equivalent in Visual Basic, it is unique to Visual LISP.

Arguments:

Program-ID – A string that names the application object class identifier. For example “Word.Application” or “Excel.Application”.

Example:

```
(setq xlapp (vlax-get-or-create-object “Excel.Application”))
```

Returns a vla-object to the external Excel application session if successful, otherwise it returns nil.

(vlax-write-enabled-p *object*)

Returns T if *object* can be modified, otherwise it returns *nil*.

Note: Be careful about this function. It may often return False when the object is in fact open for modification.

Arguments:

Object – Any vla-object

(vlax-object-erased-p *object*)

Returns T if *object* was erased from the drawing, otherwise returns *nil*.

Arguments:

Object – Any vla-object representing an Entity object type.

(vlax-release-object *object*)

Releases object from memory. Does not deallocate memory. When releasing an object that points to an external application session it is strongly suggested that `(gc)` be forced to release the external process from operating system resources.

Arguments:

Object – Any vla-object.

Warning!

While object symbols may be localized, completion of a function does not necessarily release the objects’ resources. It is advised that you still use this function to ensure the object is released properly when it is no longer needed. However, be aware that even releasing an object derived from an external application may not fully release it from memory or from the process stack of the operating system. It is best to follow the completion of your code with releasing of all unused objects, and then you should call the (gc) function in order to ”force” a garbage collection of the memory heap.

Chapter 8 –File and Directory Functions

Some of the most useful functions provided by Visual LISP are the file and directory functions. These are a collection of functions that enable you to access, and modify file properties as well as list files and folders within specified folders. One example of putting these to use, is within the context of a dialog box listbox.

Maybe you'd like to show a list of drawing files in a listbox but not show their extensions (possibly to keep the names shorter). This can be done by combining a directory listing and the vl-filename-base function in unison as follows:

```
(mapcar 'vl-filename-base (vl-directory-files pathname "*.dwg"))
```

This will return a list of names such as ("drawing1" "drawing2" …). Be careful with this example in that it provides no error checking. If the (vl-directory-files) function returns nil, the rest of the expression would crash with an error. This example is only shown to demonstrate how these functions can be combined and used to facilitate file and directory information usage.

(vl-file-size *filename*)

Returns the byte size of *filename* in integer form. If filename is not found, returns *nil*.

Arguments:

Filename	String name of file to query.

Example:

```
(vl-file-size "c:\\myfile1.txt"); returns 125523 (roughly 124 Kb)
```

(vl-file-copy *source-filename target-filename [append]*)

Copies file from source location *source-filename* to destination *target-filename*. If *append* is non-nil and destination file exists, the source file is appended to the existing destination file. If destination file exists and *append* is nil, the file will not be copied and the return value is nil. If successful, an integer value is returned.

Arguments:

Source-filename	Name of file to be copied. If file is not in the default search path, then the filename must include the full path location.
Target-filename	Name of destination to copy source file to. If destination path is not specified, the default working directory location is used.

Append	(Optional) if non-nil, indicates source file is to be appended onto destination file if destination file exists.

Examples:

```
(vl-file-copy "c:\\myfile1.txt" "c:\\mycopy.txt")

(vl-file-copy "c:\\myfile2.txt" "c:\\mycopy.txt" T); appends target file
```

(vl-file-delete *filename*)

Deletes filename. Returns T if successful, otherwise returns *nil*.

Arguments:

Filename	String name of file to delete.

(vl-file-rename *old-name new-name*)

Renames existing file from *old-name* to *new-name*. Returns T if successful, otherwise returns *nil*.

Arguments:

Old-Name	String name of existing file.
New-Name	String name to rename file when completed.

(vl-file-directory-p *filename*)

Returns T if *filename* represents a directory folder name. Returns *nil* if *filename* is actually a file or does not exist at all.

(vl-file-systime *filename*)

Returns list of date and time values for last modification of filename. Return list is in the form of (year month day-of-week day-of-month hours minutes seconds)

(vl-filename-base *filename*)

Returns the base filename without its path or extension.

Arguments:

Filename	String naming the file, with or without the path or extension.

Examples:

```
(vl-filename-base "c:\\myfiles\\drawing1.dwg")
```
Returns "drawing1"

```
(vl-filename-base "drawing1.dwg")
```
Returns "drawing1"

(vl-filename-directory *filename*)

Returns the directory or path prefix value from the specified *filename* string.

Arguments:

Filename String naming the file including the pathname.

Examples:

```
(vl-filename-directory "c:\\dwgfiles\\working\\drawing1.dwg")
Returns: "c:\\dwgfiles\\working"
```

(vl-filename-extension *filename*)

Returns the extension of a given filename string.

Arguments:

Filename Name of file (string)

Examples:

```
(vl-filename-extension "c:\\myfiles\\drawing1.dwg")
```
Returns "dwg"

(vl-filename-mktemp *[pattern directory extension]*)

Creates a unique file name to be used for a temporary file. Returns a string file name, in the format: *directory*\base<*XXX*><.*extension*> where *base* is up to 5 characters, taken from *pattern*, and *XXX* is a 3 character unique combination.

All file names generated by vl-filename-mktemp during a VLISP session are deleted when you exit VLISP.

Arguments:

Pattern A string containing a file name pattern; if nil or absent, vl-filename-mktemp uses "$VL~~".

Directory A string naming the directory for temporary files; if nil or absent, vl-filename-mktemp chooses a directory in the following order:

- The directory specified in *pattern*, if any.
- The directory specified in the TMP environment variable.
- The directory specified in the TEMP environment variable.

- The current directory.

Extension A string naming the extension to be assigned to the file; if nil or absent, vl-filename-mktemp uses the extension part of *pattern* (which may be an empty string).

Examples:

```
(vl-filename-mktemp)
"C:\\TMP\\$VL~~004"
(vl-filename-mktemp "myapp.del")
"C:\\TMP\\MYAPP005.DEL"
(vl-filename-mktemp "c:\\acad2002\\myapp.del")
"C:\\ACAD2002\\MYAPP006.DEL"
(vl-filename-mktemp "c:\\acad2002\\myapp.del")
"C:\\ACAD2002\\MYAPP007.DEL"
(vl-filename-mktemp "myapp" "c:\\acad2002")
"C:\\ACAD2002\\MYAPP008"
(vl-filename-mktemp "myapp" "c:\\acad2002" ".del")
"C:\\ACAD2002\\MYAPP00A.DEL"
```

(vl-directory-files *path pattern [mode]*)

Returns a list of files or sub-folders depending upon mode.

Arguments:

Path String name of path to query.

Pattern String denoting files to query, may contain wildcards. If not specified or nil, assumes "*.*".

Mode (Optional) Integer. One of the following…

-1 =	List directory names only
0 =	List files and directories (default if not specified)
1 =	List files only.

Examples:

```
Command: (vl-directory-files “c:\\dwgfiles\\Working” “*.dwg”)
(“drawing1.dwg” “drawing2.dwg” . . .)

Command: (vl-directory-files “c:\\dwgfiles” nil -1)
(“.” “..” “Finished” “Working”)

Command: (vl-directory-files “c:\\dwgfiles” nil 1)
nil
```

Chapter 9 –Mapping and Iteration Functions

AutoLISP provides many powerful mapping and iteration functions such as `(while)` `(foreach)` `(mapcar)` and `(apply)`. Visual LISP adds a few more that are more suited to working with ActiveX collection objects. These include `(vlax-for)`, `(vl-every)` and `(vlax-map-collection)` to name a few.

(vlax-map-collection *object function*)

Applies function over collection object members (objects). If object is not a collection, an error is generated.

Arguments:

Object	A vla-object representing a collection
Function	A symbol or lambda expression to be applied to object

Examples:

```
(setq docs (vla-get-documents (vlax-get-acad-object)))
(vlax-map-collection docs ‘vlax-dump-object)
```

This will repeat the full property listing for each document currently opened…

```
; IAcadDocument: An AutoCAD drawing
; Property values:
;   Active (RO) = -1
;   ActiveDimStyle = #<VLA-OBJECT IAcadDimStyle 046bb644>
;   ActiveLayer = #<VLA-OBJECT IAcadLayer 046bbd84>
;   ActiveLayout = #<VLA-OBJECT IAcadLayout 046b8a64>
;   ActiveLinetype = #<VLA-OBJECT IAcadLineType 046b89b4>
… cont’d…
```

(vlax-for *symbol collection [expression1 [expression2]]…*)

Iterates member objects of collection and performs expressions on each member object. If second argument is not a collection object, an error is generated. Reference to *symbol* is localized and temporary, just as with (foreach).

Arguments:

Symbol	A symbol to be assigned to each vla-object in a collection.
Collection	A vla-object representing a collection
Expressions	One or more expressions to be evaluated (optional)

Examples:

```
(setq acad (vlax-get-acad-object))
```

```
(setq layers (vla-get-layers (vla-get-activedocument acad)))
(vlax-for eachLayer layers
  (princ (vla-get-name eachLayer))
  (terpri)
)
```

This will list the names of all layers in the active drawing at the command prompt.

(vl-position *item list*)

Returns the nth position of *item* within *list* if found. If *item* is not found in *list*, returns *nil*. The position index of the first member is zero (0).

Arguments:

Item Any symbol or value.

List A list of values or symbols.

Example:

```
(setq mylist '("A" "B" "C"))
(vl-position "B" mylist) returns 1
(vl-position "b" mylist) returns nil.
```

(vl-every *predicate-function list [list]*...)

The vl-every function passes the first element of each supplied list as an argument to the test function, followed by the next element from each list, and so on. Evaluation stops as soon as one of the lists runs out.

Arguments:

Predicate-function The test function. This can be any function that accepts as many arguments as there are lists provided with vl-every, and returns T on any user-specified condition. Returns T, if *predicate-function* returns a *non-nil* value for every element combination, otherwise it returns *nil*.

The *predicate-function* value can take one of the following forms:

- A symbol (function name)
- (function (lambda (*A1 A2*) ...))

List(s) The list to be tested.

Examples:

```
Check for files larger than 1024 bytes in given folder:
```

```
(vl-every
  (function
    (lambda (filename)
      (> (vl-file-size filename) 1024)
    )
  )
  (vl-directory-files nil nil 1)
)
T
```

```
Comparing two lists...
```

```
(vl-every '= '(1 2) '(1 3))
Returns nil
(vl-every '= '(1 2) '(1 2 3))
Returns T
```

The first expression returned *nil* because vl-every compared the second element in each list and they were not numerically equal. The second expression returned T because `vl-every` stopped comparing elements after it had processed all the elements in the shorter list (1 2), at which point the lists were numerically equal. If the end of a list is reached, `vl-every` returns a non-*nil* value.

```
_$ (setq list1 (list 1 2 3 4))
(1 2 3 4)
_$ (setq list2 nil)
nil
_$ (vl-every '= list2 list1)
T
```

The return value is T because `vl-every` responds to the *nil* list as if it has reached the end of the list (even though the predicate hasn't yet been applied to any elements). And since the end of a list has been reached, `vl-every` returns a non-*nil* value.

Chapter 10 –Working with Namespaces

Developing Visual LISP VLX applications can include the use of Separate Namespaces in order to provide added performance controls and better security. However, there are some added costs in terms of coding changes that must be used to avoid problems and provide proper results. This includes importing and exporting functions and symbols, as well as passing values in and out of the local namespace.

Because a separate namespace VLX application can be isolated, it can also be queried and unloaded if desired, unlike normal LISP functions which are loaded into the document namespace and are not identifiable or capable of being unloaded by name. This very much like ObjectARX applications, and it provides added capabilities to the developer that were not available to LISP until Visual LISP came along.

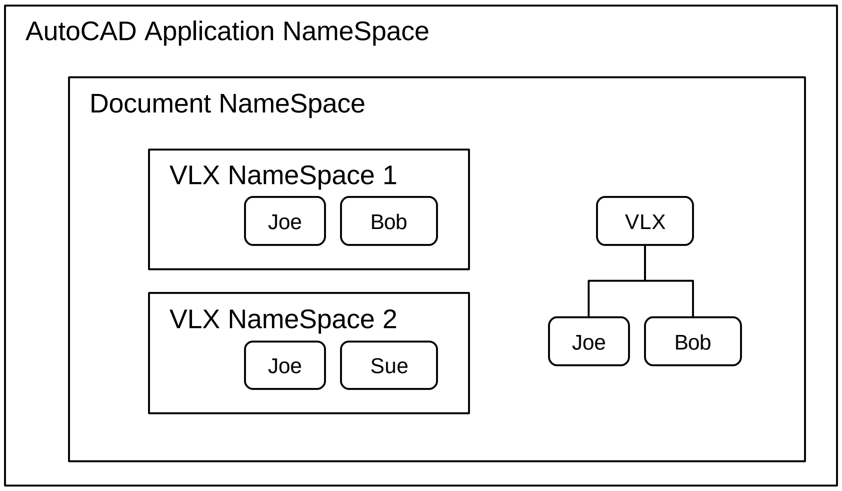


Figure 10-1 – Namespace relationships

In the world of ActiveX or COM (Component Object Model) development, every application is normally run in its own namespace within Windows. This is a common part of how multitasking is enabled. Other processes that are started by a given application may or may not run within the application's namespace. They may in fact run in their own isolated namespaces. The advantages are many, but there are tradeoffs as well.

Referring to Figure 10-1 we can use some people examples to describe how namespaces work and how processes within them behave. The two VLX applications running inside of the Document namespace are each running in their own separate namespaces. This is a little misleading, as they are not really running within the Document namespace, but are actually running within the AcadApplication namespace. Because they were loaded into the Document namespace however, they are referenced only within that Document namespace. The third VLX is not a separate-namespace application and is running entirely within the Document namespace as would any traditional AutoLISP application.

Namespace Scoping

Note in this example, that there are three objects (function definitions) named Bob. While each is loaded into the same Document namespace, they cannot see or affect each other. This results in something like having three distinct Bob objects, sort of like Bob Smith, Bob Jones and Bob Doe. Unless you incorporate some specific Visual LISP functions, they cannot communicate with each other or affect each other at all. So any objects that refer to a Bob object in this Document will only get access to the one that is in their same namespace.

The same is true for global symbols, if any are used. If we set a symbol G$BOB to a value of "A" from the command line in the Document session, any G$BOB symbols running within VLX1 or VLX2 will not be affected. From a function within VLX1, we could assign G$BOB to a value of "B". If a function within VLX1 displays (princ G$BOB) it will return "B", but from the command prompt a request for (princ G$BOB) will still return "A".

This type of protection results in what are commonly called *private functions* or *private symbols*, since they are *private* to that VLX namespace. Functions and symbols defined in the Document namespace however are not private since they are accessible by all other applications running in that namespace. To put it more accurately, private and public are relative to where the calling process is located (inside or outside of the respective namespace). In other words, object Sue is public to any functions defined and running within the VLX2 namespace, but Sue is considered private in the sense that VLX1 and other namespaces cannot access it by default.

Namespace Functions

When you intend to compile your LSP code into separate namespace VLX modules you need to make use of some special functions to get your code to communicate with other VLX modules that run outside the namespace of your new VLX module. This is true of whether the other VLX modules are within the document namespace or are compiled into their own namespaces respectively. However, it bears noting that for each VLX that is compiled into its own separate namespace, that you need to rely upon these functions in every file, not just some, at least if you need them to communicate with each other or with the document session namespace.

(vl-list-loaded-vlx)

Returns a list of all loaded separate-namespace VLX applications. If none are loaded this returns *nil*.

(vl-unload-vlx *appname*)

Unloads a separate-namespace VLX application by name (*appname* is a string value). This works like the `(arxunload)` function does with ObjectARX applications.

Arguments:

AppName	A symbol or string representing the string name of the VLX application to be unloaded, for example "myapp.vlx".

`(vl-vlx-loaded-p` *`appname`*`)`

Returns T if the specified separate-namespace VLX application is loaded in the current drawing session. Otherwise, it returns *nil*.

`(vl-doc-export` *`'function`*`)`

Exposes a function from within a separate namespace VLX application for use by applications or functions outside of its namespace. This must be declared at the top of a given LSP file, above any function definitions, prior to compiling into a separate namespace VLX application. Functions that are not exported from a given VLX are private to that VLX and cannot be accessed from outside of its namespace.

`(vl-doc-import` *`filename ['function]`*`)`

Imports a function from another VLX application for use within the current separate-namespace VLX application. If you don't import such functions that are exposed from other VLX applications, they are not accessible within a VLX application acting as a consumer of that function. If *filename* is specified, but 'function is omitted, all functions from the VLX module (*filename*) are imported.

Arguments:

Filename	A symbol or string representing a VLX filename.
Function	(Optional) A quoted symbol representing a function name.

If you want to limit the functions being imported, you must use the function argument to name those functions, one at a time. The filename argument does not use a file extension, only the base filename of the external VLX application file, and that file must reside in the default search path, or the full path and filename must be specified.

`(vl-arx-import` *`['function | "appname" ]`*`)`

Imports a function or group of functions from a specified loaded ARX file. If function and appname are omitted, all ARX-defined functions from the current document namespace are imported. This function should be used within a (defun) function definition. The ARX application must be loaded into the current document session in order for this function to work.

Arguments:

Function (Optional) A quoted symbol representing a function name.

AppName (Optional, but must be included when Function is used) A symbol or string that represents the ARX file to be imported from.

For example, if you wanted to use the DOSlib ARX function (dos_getstring) for use within your separate-namespace VLX application, you would have to import it as follows:

```
(vl-arx-import ‘dos_getstring “doslib2k.arx”)
```

If you wanted to import all functions from doslib2k.arx, you would simply leave off the function name as follows:

```
(vl-arx-import “doslib2k.arx”)
```

(vl-doc-set ‘*symbol value*)

Sets a symbol in the document namespace from within a separate-namespace VLX application. If used outside of a separate-namespace VLX application, this behaves like the (set) function. This function can be used to copy a symbol defined within a separate-namespace VLX application to the document namespace for public access. The symbol is copied by value, not by reference, meaning that the symbol within the VLX application cannot be modified from the document namespace. To import a document namespace symbol, you must use the (vl-doc-ref) function from within your separate-namespace VLX application.

Arguments:

Symbol A quoted symbol name.

Value Any value to assign to the symbol.

Examples:

Defined example function, compile into separate-namespace VLX and load into AutoCAD:

```
(defun DOCSET ()
  (vl-doc-set ‘G$NAME1 “Joe”)
)
```

From the document namespace, via the command prompt:

```
Command: (DOCSET)
Command: !G$NAME1
“Joe”
```

(vl-doc-ref ‘*symbol*)

Imports a symbol from the document namespace into a separate-namespace VLX application namespace. The symbol is copied by value, not by reference, meaning that the document namespace symbol cannot be directly modified from within the VLX namespace. To export or set a document namespace symbol from within the VLX namespace, you must use the (vl-doc-set) function.

Arguments:

Symbol	A quoted symbol name.

`(vl-load-all` ***`filename`*****`)`**

Loads a named VLX file into all opened documents at the same time. It also loads into any documents opened thereafter within the same AutoCAD application session.

Arguments:

Filename	A symbol or string representing a valid filename.

`(vl-propagate` ***`‘symbol`*****`)`**

Copies a symbol and its associated value to all opened documents within the AutoCAD application namespace, and to all documents opened afterwards during the same AutoCAD session.

Arguments:

Symbol	A quoted symbol name.

`(vl-bb-set` ***`‘symbol`*****`)`**

Posts a symbol and its associated value to the blackboard namespace. The blackboard namespace is part of the AcadApplication namespace and is accessible by all opened documents in the Documents collection. This provides similar functionality to the Windows Clipboard, except that it is used for posting and retrieving LISP symbols only.

Arguments:

Symbol	A quoted symbol name.

`(vl-bb-ref` ***`‘symbol`*****`)`**

Retrieves a symbol and its associated value from the blackboard namespace.

Arguments:

Symbol	A quoted symbol name.

(vl-list-exported-functions)

Returns a list of all functions that have been exposed to the document namespace from any loaded VLX applications.

(vlax-add-cmd *"globalname" 'function ["localname" | flags]*)

Defines a command-line function from a (defun) that is not defined as a C: function within a VLX application. You must specify at least the globalname and function options. The localname and flags options are optional. You cannot use (vlax-add-cmd) to expose functions as commands that create reactor objects or serve as reactor callbacks. Returns the *globalname* value if successful, otherwise returns *nil* if not successful.

It is suggested that (vlax-add-cmd) be used within a separate-namespace VLX and that you load the VLX using the APPLOAD command instead of from within a LISP startup routine.

Arguments:

GlobalName	A string that specifies the command name for use at the command prompt.
Function	A quoted symbol representing a function name.
LocalName	(Optional) Command name internal to the VLX application namespace. If omitted, defaults to GlobalName.
Flags	(Optional) Modify the behavior of the command with respect to transparency, pickset and pickfirst options and so forth.

Primary Flag Options:

ACRX_CMD_MODAL (0) – Command cannot be invoked while another command is active.

ACRX_CMD_TRANSPARENT (1) – Command can be invoked while another command is active.

Secondary Flag Options:

ACRX_CMD_USEPICKSET (2) – When the Pickfirst set is retrieved, it is cleared within AutoCAD. Command will be able to retrieve the Pickfirst set. Command cannot retrieve or set Grips.

ACRX_CMD_REDRAW (4) – When the Pickfirst set or grip set is retrieved, neither will be cleared within AutoCAD. Command can retrieve the Pickfirst set and the Grip set.

If both ACRX_CMD_USEPICKSET and ACRX_CMD_REDRAW are set, the effect is the same as if just ACRX_CMD_REDRAW is set. For more information on these flag options, refer to the *Command Stack* topic in the *ObjectARX Reference* manual.

Examples:

Function defined in Transparent.VLX and loaded into AutoCAD:

```
(vl-load-com)
(vl-doc-export 'example1)
(defun example1 ()
  (princ "\nThis is an example transparent function.")
  (princ)
)
(vlax-add-cmd "example1" 'example1 "example1" ACRX_CMD_TRANSPARENT)
(princ)
```

```
Command: LINE
Specify first point: 'EXAMPLE1
This is an example transparent function.
Resuming LINE command.
Specify first point:
```

(vlax-remove-cmd "*globalname*")

Removes a command definition that was previously defined using (vlax-add-cmd). The function is not affected, but the command-prompt interface is removed from the commands group.

Arguments:

GlobalName A string naming a command to be removed.

Examples:

```
(vlax-remove-cmd "example1")
T
(vlax-remove-cmd "example2")
nil
```

(vl-acad-defun '*function*)

Enables a (defun) LISP function to be used as a c: function from an ObjectARX application. This makes the function accessible to ObjectARX applications.

Arguments:

Function A quoted symbol representing a function name.

Examples:

```
(vl-acad-defun 'example1)
```

(vl-acad-undefun ‘*function*)

Undefines a command that was previously exposed using the (vl-acad-defun) function. Returns T if successful, otherwise returns *nil*.

Arguments:

Function A quoted symbol representing a function name.

Examples:

```
(vl-acad-undefun ‘example1”)
T
```

Chapter 11 –Registry Functions

Visual LISP provides special functions for accessing and modifying the Windows registry. You can query and modify keys within the HKEY_LOCAL_MACHINE, and HKEY_CURRENT_USER hives of the local registry using these functions. You cannot obtain remote registry access using Visual LISP registry functions. Nor can you access the HKEY_USERS, HKEY_CLASSES_ROOT or HKEY_CURRENT_CONFIG registry hives from Visual LISP.

Note that even in registry hives that Visual LISP can access, you are still bound by the access controls imposed by the security context of the process owner. In other words, if the Visual LISP application is being executed by a user that has limited permissions on that machine, some registry keys may not be accessible or may not be modifiable by Visual LISP. This issue is particularly important to consider in networked environments where *group policies* are used to modify registry access permissions.

(vl-registry-read *regkey [value-name]*)

Returns the value assigned to either an explicit registry key or a registry value-name (symbol) if defined in the registry. If no such registry key or value-name is found, the result is *nil*.

Arguments:

RegKey Name of a registry key in HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER hives.

Value-Name Name of a subordinate value symbol beneath the specified registry key. (Optional)

Examples:

```
(vl-registry-write “HKEY_CURRENT_USER\\Example1” “FOO” “123”)
“123”
 (vl-registry-read "HKEY_CURRENT_USER\\Example1" “FOO”)
“123”
(vl-registry-read “HKEY_CURRENT_USER\\Example1”)
nil
(vl-registry-write "HKEY_CURRENT_USER\\Example2" "" "ABCDEF")
"ABCDEF"
(vl-registry-read "HKEY_CURRENT_USER\\Example2")
"ABCDEF"
```

(vl-registry-write *regkey [value-name] value*)

Writes value to registry key or registry key value-name and returns value if successful. Returns nil if not successful.

Arguments:

RegKey	Name of registry key
Value-Name	Name of a subordinate value symbol beneath the specified registry key. (Optional)
Value	Value to write to the named registry key or value-name.

Examples:

```
(vl-registry-write “HKEY_CURRENT_USER\\Example1” “TEST1” “123”)
“123”
(vl-registry-write “HKEY_CURRENT_USER\\Example1” “” “456”)
“456”
```

(vl-registry-delete regkey [value-name])

Deletes a registry key and its associated values from the specified location in the registry. Returns T if successful, nil if it fails. If *value-name* is supplied and is not nil, the specified value will be purged from the registry. If *value-name* is absent or nil, the function deletes the specified key and all of its values. If any sub-keys exist, regkey cannot be deleted. To remove keys that have sub-keys, you must use `(vl-registry-descendents)` to collect the sub-keys and delete them first.

Arguments:

RegKey	Name of registry key
Value-Name	Name of a subordinate value-name (symbol) beneath the specified registry key. (Optional)

Examples:

```
(vl-registry-write "HKEY_CURRENT_USER\\Example1" "TEST1" "123")
"123"
(vl-registry-delete "HKEY_CURRENT_USER\\Example1")
T
```

(vl-registry-descendents *regkey [value-names]*)

Returns a list of subkeys or value-names beneath the specified registry key. If *value-names* is supplied and is not nil, the specified value names will be listed from the registry. If *value-name* is absent or nil, the function displays all subkeys of *regkey*. Note also that the return value is often in reverse sorted order.

Arguments:

RegKey	Name of registry key
Value-Names	A string containing the values for the *regkey* entry.

Examples:

```
(vl-registry-descendents "HKEY_LOCAL_MACHINE\\SOFTWARE")

("WexTech Systems" "Voice" "Synaptics" "Symantec" "Secure"
"Program Groups" "Policies" "ODBC" "Nico Mak Computing"
"MicroVision" "Microsoft" "MetaStream" "McNeel" "McAfee"
"JavaSoft" "Intel Corporation" "INTEL" "InstalledOptions"
"Helios" "DOSLib" "Dell Computers" "Dell Computer Corporation"
"Dell Computer" "DameWare Development" "Clients" "Classes" "BVRP
Software" "BigFix" "Autodesk" "ATI Technologies" "Apple Computer,
Inc." "America Online" "Adobe" "Adaptec" "3Com")
```

You can see more examples of registry functions by opening the RegDump.LSP file located in the Visual LISP Samples directory of your AutoCAD 2002 installation. In this file, you will find a useful function named `(registry-tree-dump)` that performs a recursive search of all subkeys and value-names beneath a specified registry key.

Tip!

You can create a pair of Get and Set functions to store and retrieve registry values with some control over standardized locations and error trapping. You might find the following two functions helpful:

```
(setq G$REGROOT "HKEY_CURRENT_USER\\Software\\MyApplication\\")

(defun RegGet (key default / val)
  (if (= nil (setq val (vl-registry-read (strcat G$REGROOT key))))
    (progn
      (regset key default)
      (setq val (vl-registry-read (strcat G$REGROOT key)))
    )
  )
  (if val val default)
)

(defun RegSet (key val)
  (vl-registry-write (strcat G$REGROOT key) ""  val)
)
```

Chapter 12 – Reactors and Call-Backs

Note: Parts of this section were derived from the AutoCAD 2002 online Help documentation with some modifications to provide additional examples or clarity.

Warning!

While reactors are indeed extremely powerful and useful to developers, they should be used with careful moderation. Depending upon the types and volume of reactors you define in a given situation, you can easily and quickly deplete system resources and cause AutoCAD to become unresponsive and even unstable or crash. Be careful in choosing how you will apply reactors to your applications development.

Reactors are simply links between AutoCAD and your applications that allow you to make functions that respond to *events* that occur within AutoCAD. For example, you can create a reactor to notify your applications that an entity was erased. The application could then perform some action in response to this event. A button on a form is a simple example of event-driven programming that uses an Event and a Response to perform an action. When you pick the button, it fires an event, much like a signal or broadcast. This event is detected by a Reactor of sorts that performs some action as a result by using what is known as a *call-back* process.

In AutoCAD terms, you might consider a scenario such as using the `CommandWillStart` event to fire a Command Reactor call-back to perform some action based on what command was executed. Maybe the user initiated the HATCH command, and you want to react to that by firing a call-back function that sets a special layer active before the Hatch is placed in the drawing, and then restore the previous layer state when the command finishes. Maybe you'd like it to also restore the previous layer state if the command fails due to an error? Or what if the user simply cancels the command in midstream? This is possible using Reactors and Visual LISP programming.

What you need to do first is define the function that will be used in the call-back. Be careful not to use `(command)` or `(vl-cmdf)` anywhere in that function if it will be called upon as the result of a Command Reactor as this may start an endless cycle and crash AutoCAD. Sounds like common sense, huh? Sometimes things like this are not so obvious and can cause big problems. This is but one reason for developers to be VERY careful when considering Reactors.

The next thing you need to do is define the Reactor and construct it to call your call-back function if the proper condition is met by the event detected (the command is HATCH or BHATCH, ignoring all others).

Figure 12-1 demonstrates how you might use a Command Reactor to respond to the HATCH or BHATCH command by defining and setting layer "HATCHING" active until the command has either finished (via the `CommandEnded` event), or aborted due to error (via the `CommandFailed` event), or user cancellation (via the `CommandCancelled` event).

Visual LISP Reactor Functions

(vl-load-com)	Loads AutoLISP reactor support functions and other AutoLISP extensions
(vlr-acdb-reactor *data callbacks*)	Constructs a global "database" reactor object
(vlr-add *obj*)	Enables a disabled reactor object
(vlr-added-p *obj*)	Tests to determine whether a reactor object is enabled
(vlr-beep-reaction [*args*])	Produces a beep sound
(vlr-current-reaction-name)	Returns the name (symbol) of the current event, if called from within a reactor's callback
(vlr-data *obj*)	Returns application-specific data associated with a reactor
(vlr-data-set *obj data*)	Overwrites application-specific data associated with a reactor
(vlr-deepclone-reactor *obj data*)	Constructs an editor reactor object that notifies of deep clone events
(vlr-docmanager-reactor *obj data*)	Constructs a reactor object that notifies of MDI-related events
(vlr-dwg-reactor *obj data*)	Constructs an editor reactor object that notifies of a drawing event (for example, opening or closing a drawing file)
(vlr-dxf-reactor *obj data*)	Constructs an editor reactor object that notifies of an event related to reading or writing of a DXF file
(vlr-editor-reactor *data callbacks*)	Constructs a global "editor" reactor object
(vlr-linker-reactor *data callbacks*)	Constructs a global "linker" reactor object
(vlr-miscellaneous-reactor *data callbacks*)	Constructs an editor reactor object that does not fall under any of the other editor reactor types
(vlr-mouse-reactor *data callbacks*)	Constructs an editor reactor object that notifies of a mouse event (for example, a double-click)
(vlr-notification *reactor*)	Determines whether or not a reactor's callback function will execute if its associated namespace is not active
(vlr-object-reactor *owners data callbacks*)	Constructs an object reactor object
(vlr-owner-add *reactor owner*)	Adds an object to the list of owners of an object reactor
(vlr-owner-remove *reactor owner*)	Removes an object from the list of owners of an object reactor
(vlr-owners *reactor*)	Returns the list of owners of an object reactor

(vlr-pers *reactor*)	Makes a reactor persistent between sessions (*not* transient)
(vlr-pers-list [*reactor*])	Returns a list of persistent reactors in the current drawing
(vlr-pers-p *reactor*)	Determines whether or not a reactor is persistent
(vlr-pers-release *reactor*)	Makes a reactor transient (*not* persistent)
(vlr-reaction-name *reactor-type*)	Returns a list of all callback conditions for this reactor type
(vlr-reaction-set *reactor event function*)	Adds or replaces a callback function in a reactor
(vlr-reactions *reactor*)	Returns a list of pairs (*event-name . callback_function*) for the reactor
(vlr-reactors *[reactor-type...]*)	Returns a list of reactors of the specified types
(vlr-remove *reactor*)	Disables a reactor object
(vlr-remove-all *reactor-type*)	Disables all reactors of the specified type
(vlr-set-notification *reactor `range*)	Defines whether or not a reactor's callback function will execute if its associated namespace is not active
(vlr-sysvar-reactor *data callbacks*)	Constructs an editor reactor object that notifies of a change to a system variable
(vlr-toolbar-reactor *data callbacks*)	Constructs an editor reactor object that notifies of a change to the bitmaps in a toolbar
(vlr-trace-reaction)	A pre-defined callback function that prints one or more callback arguments in the Trace window
(vlr-type *reactor*)	Returns a symbol representing the reactor type
(vlr-types)	Returns a list of all reactor types (see next section)
(vlr-undo-reactor *data callbacks*)	Constructs an editor reactor object that notifies of an undo event
(vlr-wblock-reactor *data callbacks*)	Constructs an editor reactor object that notifies of an event related to writing a block
(vlr-window-reactor *data callbacks*)	Constructs an editor reactor object that notifies of an event related to moving or sizing an AutoCAD window
(vlr-xref-reactor *data callbacks*)	Constructs an editor reactor object that notifies of an event related to attaching or modifying XREF

Reactor Types

There are many types of AutoCAD reactors. Each reactor type responds to one or more AutoCAD events. The different types of reactors are grouped into one of the following five categories:

Database Reactors

Database reactors notify your application when specific events occur to the drawing database, such as when an object has been added to the drawing database.

Document Reactors

Document reactors notify your application of a change to the current drawing document, such as opening a new drawing document, activating a different document window, and changing a document's lock status. This does not include of all the events covered by Editor reactors.

Editor Reactors

Editor reactors notify you each time an AutoCAD command is invoked; a drawing is opened, closed, or is saved; a DXF file is imported or exported; or a system variable setting is modified.

Linker Reactors

Linker reactors notify your application every time an ARX application is loaded or unloaded.

Object Reactors

Object reactors notify you each time a specific object is changed, copied, or deleted.

With the exception of Editor reactors, there is one type of reactor for each reactor category. The following table lists the name by which each reactor type is identified in the Visual LISP environment:

General reactor types	
Reactor type identifier	**Description**
:VLR-AcDb-Reactor	A Database reactor
:VLR-DocManager-Reactor	A Document management reactor
:VLR-Editor-Reactor	A General Editor reactor-maintained for backward-compatibility
:VLR-Linker-Reactor	A Linker reactor
:VLR-Object-Reactor	An Object reactor

The Editor reactor class is broken down into more specific reactor types. The `:VLR-Editor-Reactor` type is retained for backward-compatibility, but any new Editor reactors introduced with AutoCAD 2000 cannot be referenced through `:VLR-Editor-Reactor`. The following table lists the types of Editor reactors available beginning with AutoCAD 2000.

Editor reactor types	
Reactor type	**Description**
:VLR-Command-Reactor	Notifies of a command event
:VLR-DeepClone-Reactor	Notifies of a deep clone event
:VLR-DWG-Reactor	Notifies of a drawing event (for example, opening or closing a drawing file)
:VLR-DXF-Reactor	Notifies of an event related to reading or writing of a DXF file
:VLR-Insert-Reactor	Notifies of an event related to block insertion
:VLR-Lisp-Reactor	Notifies of a LISP event
:VLR-Miscellaneous-Reactor	Does not fall under any of the other editor reactor types
:VLR-Mouse-Reactor	Notifies of a mouse event (for example, a double-click)
:VLR-SysVar-Reactor	Notifies of a change to a system variable
:VLR-Toolbar-Reactor	Notifies of a change to the bitmaps in a toolbar
:VLR-Undo-Reactor	Notifies of an UNDO event
:VLR-Wblock-Reactor	Notifies of an event related to writing a block
:VLR-Window-Reactor	Notifies of an event related to moving or sizing an AutoCAD window
:VLR-XREF-Reactor	Notifies of an event related to attaching or modifying XREFs

Note: Use the `vlr-types` function to return the complete list of reactor types.

For each reactor type there are a number of events that can cause the reactor to notify your application. These events are known as callback events, because they cause the reactor to call a function you associate with the event. For example, when you issue the SAVE or QSAVE commands to save a drawing, a `:vlr-beginSave` event occurs. When you complete the save process, a `:vlr-saveComplete` event occurs. In designing a reactor-based application, it is up to you to determine the events you are interested in, and to write the callback functions to be triggered when these events occur.

The `vlr-reaction-names` function returns a list of all available events for a given reactor type:

```
(vlr-reaction-names reactor type)
```

For example, the following command returns a list of all events related to Object reactors:

```
$ (vlr-reaction-names :VLR-Object-Reactor)
(:VLR-cancelled :VLR-copied :VLR-erased :VLR-unerased :VLR-
goodbye :VLR-openedForModify :VLR-modified :VLR-subObjModified
:VLR-modifyUndone :VLR-modifiedXData :VLR-unappended :VLR-
reappended :VLR-objectClosed)
```

NOTE: If this or any other `vlr-*` command fails with a "no function definition" message, you may have forgotten to call `vl-load-com`, the function that loads AutoLISP reactor support functions.

To print out a list of all available reactor events, sorted by reactor type, load and execute the following example function:

```
(defun print-reactors-and-events ()
  (foreach rtype (vlr-types)
    (princ (strcat "\n" (vl-princ-to-string rtype)))
    (foreach rname (vlr-reaction-names rtype)
      (princ (strcat "\n\t" (vl-princ-to-string rname)))
    )
  )
  (princ)
)
```

Verifying Reactor Types

The *AutoLISP Reference* lists each event available for a reactor type. For each reactor type, you can find this information by looking up the description of the function you use to define a reactor of that type. These functions have the same name as the reactor type, minus the leading colon. For example, `vlr-acdb-reactor` creates a database reactor, `vlr-toolbar-reactor` creates a toolbar reactor, and so on.

`(vlr-type` *`reactor`*`)`

Iterates member objects of collection and performs expressions on each member object. If second argument is not a collection object, an error is generated. Reference to *symbol* is localized and temporary, just as with (foreach).

Arguments:

reactor A reactor object.

Return Values

A symbol identifying the reactor type. The following table lists the types that may be returned by the vlr-type function:

Reactor types

Reactor type	Description
:VLR-AcDb-Reactor	A drawing database reactor.
:VLR-Command-Reactor	An editor reactor notifying of a command event. This does not include C: commands defined by LISP expressions, only native AutoCAD commands.
:VLR-DeepClone-Reactor	An editor reactor notifying of a deep clone event.
:VLR-DocManager-Reactor	A document management reactor.
:VLR-DWG-Reactor	An editor reactor notifying of a drawing event (for example, opening or closing a drawing file).
:VLR-DXF-Reactor	An editor reactor notifying of an event related to reading or writing of a DXF file.
:VLR-Editor-Reactor	A general editor reactor; maintained for backward-compatibility.
:VLR-Insert-Reactor	An editor reactor notifying of an event related to block insertion.
:VLR-Linker-Reactor	A linker reactor.
:VLR-Lisp-Reactor	An editor reactor notifying of a LISP event.
:VLR-Miscellaneous-Reactor	An editor reactor that does not fall under any of the other editor reactor types.
:VLR-Mouse-Reactor	An editor reactor notifying of a mouse event (for example, a double-click).
:VLR-Object-Reactor	An object reactor. (any object, e.g. vla-object type)
:VLR-SysVar-Reactor	An editor reactor notifying of a change to a system variable.
:VLR-Toolbar-Reactor	An editor reactor notifying of a change to the bitmaps in a toolbar.
:VLR-Undo-Reactor	An editor reactor notifying of an undo event.
:VLR-Wblock-Reactor	An editor reactor notifying of an event related to writing a block.
:VLR-Window-Reactor	An editor reactor notifying of an event related to moving or sizing an AutoCAD window.
:VLR-XREF-Reactor	An editor reactor notifying of an event related to attaching or modifying XREFs.

Examples

```
_$ (vlr-type circleReactor)
:VLR-Object-Reactor
```

There are various ways to obtain information about reactors. Visual LISP supplies AutoLISP functions to query reactors, and you can use standard Visual LISP data inspection tools to view information on them.

To use AutoLISP to list all reactors in a drawing, call the `vlr-reactors` function. The function returns a list of reactor lists. Each reactor list begins with a symbol identifying the reactor type, followed by pointers to each reactor of that type. For example:

```
_$ (vlr-reactors)

((:VLR-Object-Reactor #<VLR-Object-Reactor>) (:VLR-Editor-Reactor

#<VLR-Editor-Reactor>))
```

In this example, `vlr-reactors` returned a list containing two lists, one identifying a single object reactor and one identifying a single Editor reactor.

To list all reactors of a given type, supply `vlr-reactors` with an argument identifying the reactor type. Specify one of the values returned by the `vlr-types` function; these are listed in the section on Reactor Types. For example, the following lists all DWG reactors:

```
_$ (vlr-reactors :vlr-dwg-reactor)

((:VLR-DWG-Reactor #<VLR-DWG-Reactor> #<VLR-DWG-Reactor>))
```

In this case, the return value is a list containing one list. The one list identifies pointers to two DWG reactors.

Using Object Reactors

Unlike other AutoCAD reactors, object reactors are attached to specific AutoCAD entities (objects). When you define an object reactor, you must identify the entity the reactor is to be attached to. The `vlr-object-reactor` function, which creates object reactors, requires the following arguments:

- A list of VLA-objects identifying the drawing objects that are to fire notifications to the reactor. These objects are referred to as the *reactor owners.*

- AutoLISP data to be associated with the Reactor object.

- A list of pairs naming the event and the callback function to be associated with that event *(event-name . callback_function).*

Warning!

You cannot modify an object in a callback function if it is included in the object reactor's owner list. Attempts to do so will generate an error message and can crash AutoCAD. This is a very esoteric concern with using reactors: To carefully plan your implementation strategy so that you avoid the possibility of circular references where a reactor callback affects one of the sources of the reactor itself.

For example, the following statement defines an object reactor with a single owner (the object identified by myCircle), then attaches the string "Circle Reactor" to the reactor and tells AutoCAD to invoke the print-radius function when a user modifies myCircle:

```
(setq circleReactor (vlr-object-reactor (list myCircle)

         "Circle Reactor" '((:vlr-modified . print-radius))))
```

The Reactor object is stored in variable `circleReactor`; you can refer to the reactor using this variable. When defining a list of owners, you must specify VLA-objects only; Ename objects are not allowed. VLA-objects are required because callback functions can only use ActiveX methods to modify AutoCAD objects, and ActiveX methods require a VLA-object to work on.

Note that, although you cannot use objects obtained through functions such as entlast and entget with callback reactors, you can convert these Ename objects into VLA-objects using the `vlax-ename->vla-object` function. See the *AutoLISP Reference* for more information on vlax-ename->vla-object.

The following code example draws a circle and applies a reactor to the circle to notify of any change made to the entity thereafter. Load the code, draw the circle and then go back and either move or resize the circle using SCALE or grip editing to see how it works.

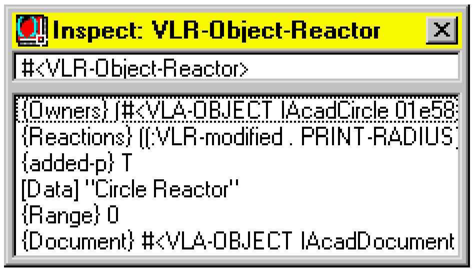
```
(vl-load-com)
(setq oAcad (vlax-get-acad-object)
      oDoc  (vla-get-activedocument oAcad)
)

(cond
  ( (and
      (setq ctrPt (getpoint “\nCenter point: “))
      (setq rad   (distance ctrPt (getpoint ctrPt “\nRadius: “)))
    )
    (setq CircleObject
      (vla-addCircle
        (vla-get-ModelSpace oDoc)
        (vlax-3d-point ctrPt)
        radius
      )
    )
  )
)

(if CircleObject
  (setq circleReactor
    (vlr-object-reactor (list CircleObject) “Circle Reactor”
      ‘((:vlr-modified . rShowRadius))
    )
  )
)

(defun rShowRadius
  (notifier-object reactor parameter-list)
  (cond
    ( (vlax-property-available-p notifier-object "Radius")
     (princ "*** The radius is ")
     (princ (vla-get-radius notifier-object))
    )
  )
)
```

Attaching Data to Reactor Objects

The object reactor creation example in Using Object Reactors included a string, "Circle Reactor," in the call to `vlr-object-reactor`. You do not have to specify any data to be included with the reactor; you can specify nil instead. However, an object may have several reactors attached to it. Include an identifying text string, or other data your application can use, to allow you to distinguish among the different reactors attached to an object.

Inspecting Reactors Within the VLIDE

You can examine reactors using the VLIDE **Inspect** tool. For example, the object reactor defined in Using Object Reactors was returned to the variable circleReactor. If you open an Inspect window for this variable, VLISP displays the following information:



```
Inspect: VLR-Object-Reactor
#<VLR-Object-Reactor>
{Owners} (#<VLA-OBJECT IAcadCircle 01e58
{Reactions} ((:VLR-modified . PRINT-RADIUS
{added-p} T
[Data] "Circle Reactor"
{Range} 0
{Document} #<VLA-OBJECT IAcadDocument
```

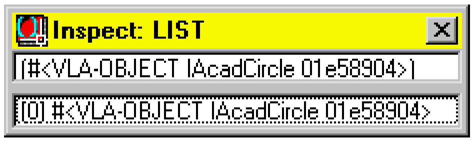
The following information is revealed in the Inspect list:

Objects owning the reactor

- Event and associated callback function
- Whether or not the reactor is active:
    - Yes if added-p is T
    - No if added-p is nil
- User data attached to the reactor
- Document range in which the reactor will fire:
    - 0 - Reactor responds only in the context of the drawing document it was created in.
    - 1 - Reactor responds in the context of any document
    - See the section “Reactors and Multiple Namespaces”

- The AutoCAD document attached to the object reactor

Double-click on the item that begins with {Owners} to view a list of the owner objects:



```
(#<VLA-OBJECT IAcadCircle 01e58904>)
[0] #<VLA-OBJECT IAcadCircle 01e58904>
```

You can drill down to find the owner by double-clicking on a list item in the Inspect list box.

Querying Reactors

VLISP also provides functions to inspect a reactor definition from within an application program, or at the Console prompt:

- vlr-type returns the type of the specified reactor. For example:

```
$ (vlr-type circleReactor)
:VLR-Object-Reactor
```

- `vlr-current-reaction-name` returns the name of the event that caused the callback function to be called.
- `vlr-data` returns the application-specific data value attached to the reactor, as shown in the following example:

```
$ (vlr-data circleReactor)
"Circle Reactor"
```

You can use this data to distinguish among multiple reactors that can fire the same callback function.

- vlr-owners returns a list of the objects in an AutoCAD drawing that fire notifications to an object reactor. The following function call lists the objects that fire notifications to circleReactor:

```
_$ (vlr-owners circleReactor)
(#<VLA-OBJECT IAcadCircle 03ad077c>)
```

- `vlr-reactions` returns the callback list of condition-function pairs of the specified reactor. The following example returns information about circleReactor:

```
$ (vlr-reactions circleReactor)
((:vlr-modified . PRINT-RADIUS))
```

Transient and Persistent Reactors

Reactors may be transient or persistent. Transient reactors are lost when a drawing closes; this is the default reactor mode. Persistent reactors are saved with the drawing and exist when the drawing is next opened. If you use persistent reactors that invoke custom applications via a callback, the custom applications must be loaded for the callback to work properly. See the next section for more on peristent reactors.

Use the `vlr-pers` function to make a reactor persistent. To remove persistence from a reactor and make it transient, use vlr-pers-release. Each function takes a Reactor object as its only argument. For example, the following command makes a reactor persistent:

```
_$ (vlr-pers circleReactor)
#<VLR-Object-Reactor>
```

If successful, `vlr-pers` returns the specified Reactor object.

To determine whether a Reactor object is persistent or transient, issue `vlr-pers-p`. For example:

```
_$ (vlr-pers-p circleReactor)
#<VLR-Object-Reactor>
```

The `vlr-pers-p` function returns the Reactor object if it is persistent, `nil` if it is not.

Opening Drawings with Persistent Reactors

Since a reactor is only a link between an event and a callback function. While this link remains, the callback function itself is not part of the reactor, and is normally not part of the drawing. The reactors saved in the drawing are only usable if their associated callback functions are loaded in AutoCAD. You can cause this to occur automatically when a drawing is opened if you define the reactor and callback functions in a separate-namespace VLX.

If you open a drawing containing VLISP reactor information and the associated callback functions are not loaded, AutoCAD displays an error message. You can use the `vlr-pers-list` function to return a list of all Persistent reactors in a drawing document.

Reactors and Multiple Namespaces

The current implementation of AutoLISP supports working in one drawing document at a time. Some AutoCAD APIs, such as ObjectARX and VBA, do support the ability of an application to work simultaneously in multiple documents. As a result, an application may modify an open drawing that is not currently active. This is not supported in AutoLISP.

Guidelines for Using Reactors

As I mentioned at the start of this chapter, Reactors demand careful attention to planning and consderation for performance and stability. The following guidelines are provided in the AutoCAD online Help documentation and are very good points to consider.

When using reactors, try to adhere to the following guidelines. Reactors that violate these guidelines can result in unpredictable results for your application if the internal implementation of reactors changes.

- Do not rely on the sequence of reactor notifications.

It is recommended that, with a few exceptions, you do not rely on the sequence of reactor notifications. For example, an OPEN command triggers BeginCommand, BeginOpen, EndOpen, and EndCommand events. However, they may not occur in that order. The only event sequence you can safely rely on is that a Begin event will occur before the corresponding End event. For example, commandWillStart() always occurs before commandEnded(), and beginInsert() always occurs before endInsert(). Relying on more complex sequences may result in problems for your application if the sequence is changed as a result of new notifications being introduced in the future and existing ones being rearranged.

- Do not rely on the sequence of function calls between notifications.

It is not guaranteed that certain functions will be called between certain notifications. For example, when you receive :vlr-erased notification on object A, all it means is that object A is erased. If you receive :vlr-erased notification on A followed by a :vlr-erased notification on B, all it means is that both objects A and B are erased; it does not ensure that B was erased after A. If you tie your application to this level of detail, there is a very high probability of your application breaking in future releases. Instead of relying on sequences, rely on reactors to indicate the state of the system.

- Do not use any *interactive* functions in your reactor callback function (for example, do not use getPoint, entsel).

Attempting to execute interactive functions from within a reactor callback function can cause serious problems, as AutoCAD may still be processing a command at the time the event is triggered. Therefore, avoid the use of input-acquisition methods such as getPoint, entsel, and getkword, as well as selection set operations and the command function.

- Do not launch a dialog box from within an event handler.

Dialog boxes are considered interactive functions and can interfere with the current operation of AutoCAD. However, message boxes and alert boxes are not considered interactive and can be issued safely.

- Do not update the object that issued the event notification.

The event causing an object to trigger a callback function may still be in progress and the object still in use by AutoCAD when the callback function is invoked. Therefore, do not attempt to update an object from a callback function for the same object. You can, however, safely read information from the object triggering an event. For example, suppose you have a floor filled with tiles and you attach a reactor to the border of the floor. If you change the size of the floor, the reactor callback function will automatically add or subtract tiles to fill the new area. The function will be able to read the new area of the border, but it cannot attempt any changes on the border itself.

- Do not perform any action from a callback function that will trigger the same event.

If you perform an action in your reactor callback function that triggers the same event, you will create an infinite loop. For example, if you attempt to open a drawing from within a BeginOpen event, AutoCAD will simply continue to open more drawings until the maximum number of open drawings is reached.

- Verify that a reactor is not already set before setting it, or you may end up with multiple callbacks on the same event.

- Remember that no events will be fired while AutoCAD is displaying a modal dialog.

Warning! A VLX may run in a separate-namespace from the document it is loaded from, but it is still associated with that originating document and cannot manipulate objects in other documents.

Visual LISP provides limited support for reactor callback functions executing in a document that is not active. By default, a reactor callback function will execute only if a notification event occurs when the document it was defined in is the active document. You can alter this behavior using the `vlr-set-notification` function.

To specify that a reactor should execute its callback function even if the document it was defined in is not active (for example, if an application in another namespace triggers an event), issue the following function call:

```
(vlr-set-notification reactor-object 'all-documents)
```

This can be useful to notify all instances of your application (that is, if not a separate namespace VLX application propagated to all sessions) that an event occurred in one of the sessions.

To modify a reactor so it only executes its callback function if an event occurs when the document it was defined in is active, issue the following:

```
(vlr-set-notification reactor-object 'active-document-only)
```

The `vlr-set-notification` function returns the specified reactor object. For example, the following sequence of commands defines a reactor and sets it to respond to events whether or not its associated document is active:

```
_$ (setq circleReactor (vlr-object-reactor (list myCircle)
"Circle Reactor" '((:vlr-modified . print-radius))))
#<VLR-Object-Reactor>
_$ (vlr-set-notification circleReactor 'all-documents)
#<VLR-Object-Reactor>
```

To determine the notification setting of a reactor, use the `vlr-notification` function. For example:

```
_$ (vlr-notification circleReactor)
all-documents
```

The `vlr-set-notification` function affects only the specified reactor. All reactors are created with the default notification set to active-document-only.

Warning!

If you choose to set a reactor to execute its callback function even if triggered when its document is not active, the callback function should do nothing other than set and read AutoLISP system variables. Performing other types of actions may cause AutoCAD to become unstable or crash!

Chapter 13 – Making Visual LISP Applications

Probably the most significant feature provided by Visual LISP is the capability to build and manage applications. In this context, we are really talking about VLX applications, but you could consider FAS output to be applications as well. The building of Visual LISP applications is closely tied to the use of Projects, but they are not inseparable.

Why Make VLX Applications?

The main reasons for making VLX applications are improved security and performance. Security is improved because the Visual LISP compiler encrypts and compiles the source LSP code into binary output that is unreadable to the human eye. It also can combine multiple LSP files into a single VLX output further improving security and providing a single output for delivery to users. Performance is improved because the compiled code is actually more efficient to execute at runtime.

Unlike many other programming languages, Visual LISP does not truly “compile” its output, but more accurately performs an encryption and partial compilation. This is somewhat like what a Java compiler does to produce “p-code” output, which is then compiled at runtime by the JVM (Java Virtual Machine) compiler on the client. Visual LISP VLX code is compiled to binary output but not to machine level language, meaning it must be interpreted at runtime by the client. It still provides some measure of performance improvement over raw LSP source code though.

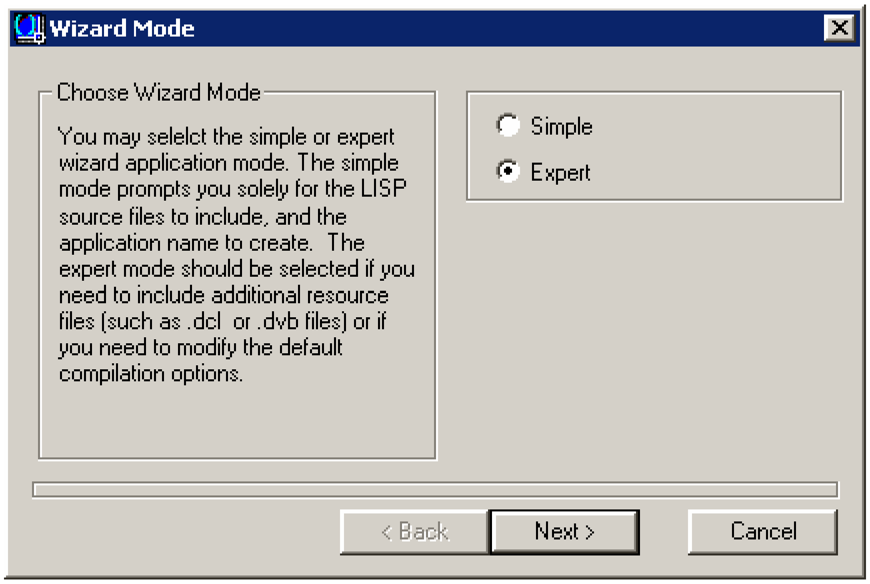
FAS files are an intermediate output during the VLX compilation that are the product of the LSP file compilation. The VLX module combines FAS files and any other file types to wrap it all up as a single loadable module on the client. VLX applications can include other file types such as LISP, Dialog Control Language files (.DCL), Compiled LISP code (.FAS), VBA compiled files (.DVB), ASCII TEXT (.TXT) and even other VLISP Projects (.PRV).

One of the most useful features of making VLX applications is that you can combine multiple files into the single VLX output. This makes for easy loading and management as well as keeping your deliverable product clean and compact. Let’s try an example.

Building a Simple Application

Open the FirstApplication.LSP file in Visual LISP from the book samples CD. Then open the FirstApplication.DCL file in Visual LISP as well. Now, pick File/Make Application/New Application Wizard from the pulldown menu. There are two modes for compiling VLX applications: Simple and Expert. Simple used when you are only going to compile LSP files and do not intend to compile a separate namespace VLX. Expert mode allows you to include additional Resource files such as DCL, DVB, VLX and others within the VLX as well as make it a separate namespace application.

Since in this exercise we will be compiling a LSP file with a DCL file into a single, separate namespace VLX application, you have to select the Expert mode from the Wizard Mode panel (Figure 13-1). Pick the Next button.

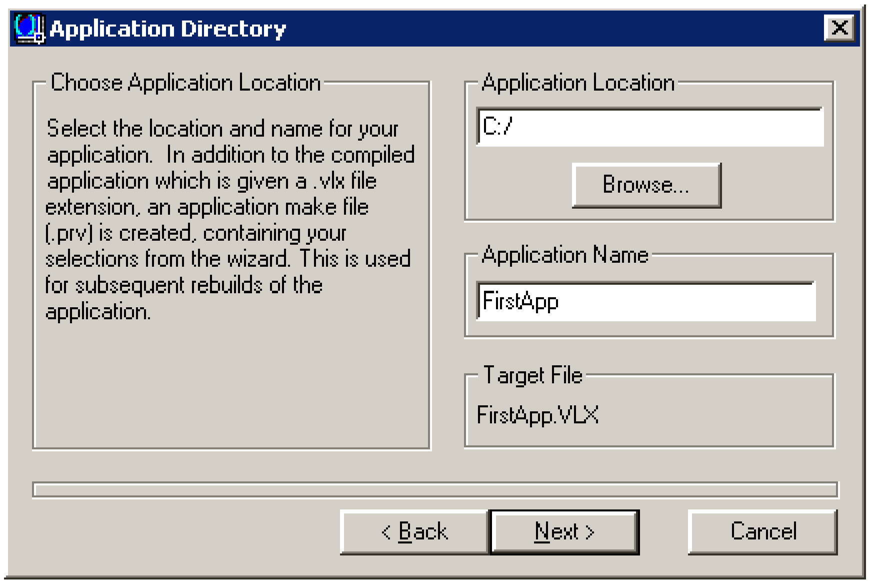


You may selelct the simple or expert wizard application mode. The simple mode prompts you solely for the LISP source files to include, and the application name to create. The expert mode should be selected if you need to include additional resource files (such as .dcl or .dvb files) or if you need to modify the default compilation options.

Simple [✓ Expert]

Figure 13-1 – Make Application Wizard

The Application Directory panel (Figure 13-2) is where you specify the VLX filename and target output location. The Application Location is where you want the VLX file to be created at the end of the process. The Application Name is the name you want to call the VLX file (don't include the extension, only the base filename). You will see that while you type in the Application Name box, the Target File window shows the actual VLX filename result. Once you've specified the Application Location, and entered the Application Name "FirstApplication", pick the Next button to continue on.



Select the location and name for your application. In addition to the compiled application which is given a .vlx file extension, an application make file (.prv) is created, containing your selections from the wizard. This is used for subsequent rebuilds of the application.

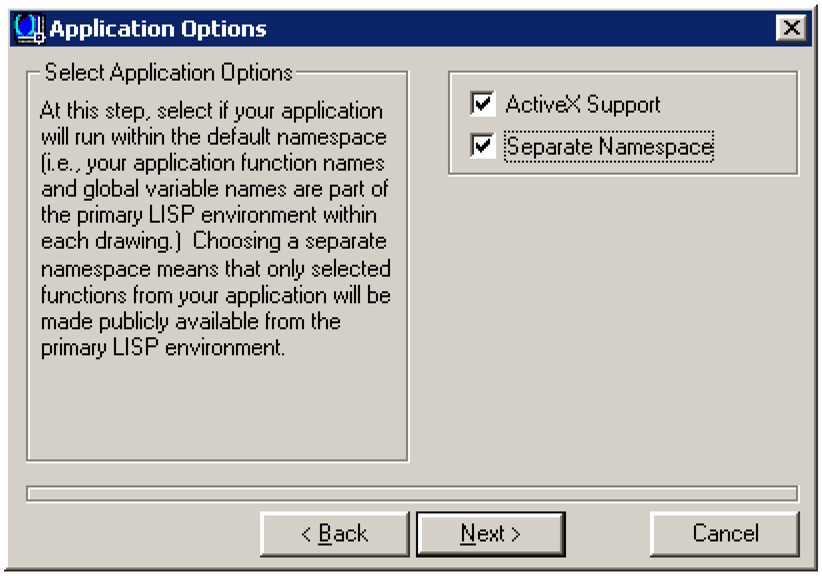
Application Location: C:/

Application Name: FirstApp

Target File: FirstApp.VLX

Figure 13-2 – Specifying the output location and VLX filename

The Application Options panel (Figure 13-3) prompts you to make this a Separate Namespace application, as well as use ActiveX Support. For this example, check both options, and pick Next to continue.


Application Options

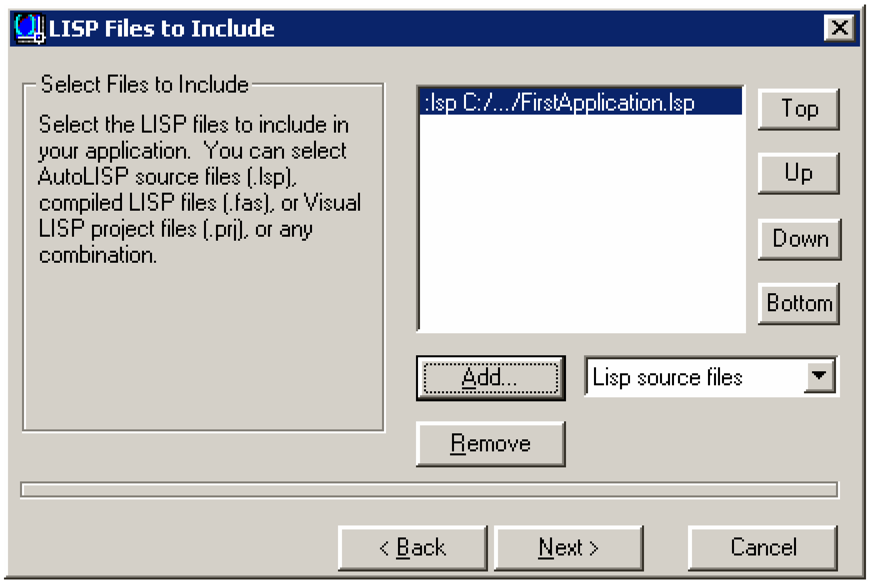
Select Application Options

At this step, select if your application will run within the default namespace (i.e., your application function names and global variable names are part of the primary LISP environment within each drawing.) Choosing a separate namespace means that only selected functions from your application will be made publicly available from the primary LISP environment.

[✓] ActiveX Support
[✓] Separate Namespace

Figure 13-3 – Application Options form (Separate Namespace checked)

The LISP Files to Include panel (Figure 13-4) is where you select the LISP code files (*.LSP) to include in your VLX application. Pick the Add button to browse for, and select the FirstApplication.LSP file. Then pick Next to continue.


LISP Files to Include

Select Files to Include

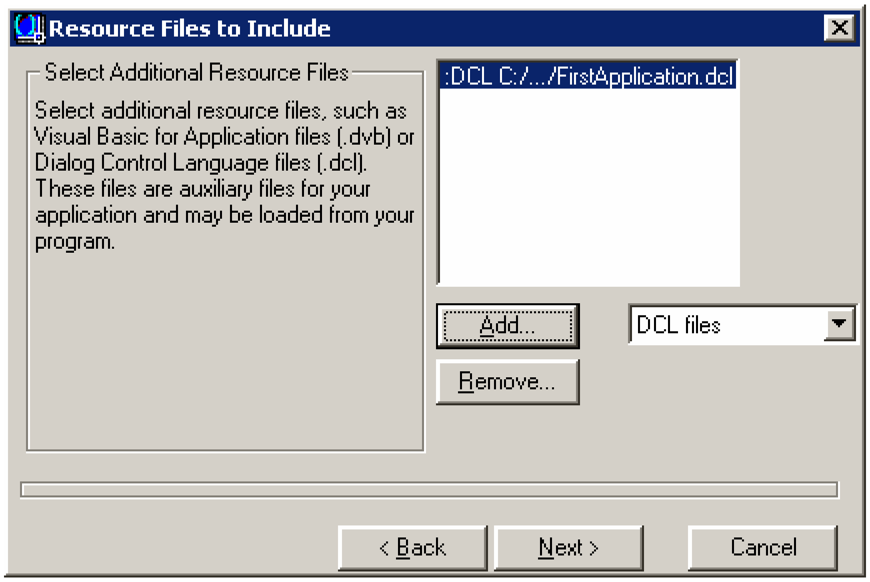
Select the LISP files to include in your application. You can select AutoLISP source files (.lsp), compiled LISP files (.fas), or Visual LISP project files (.prj), or any combination.

:lsp C:/.../FirstApplication.lsp

Lisp source files

Figure 13-4 – LISP Files to Include form

The Resource Files to Include panel (Figure 13-5) is where you select additional resource files, such as DCL dialog form files, DVB (VBA) files, and other types of files. Change the file type selection to DCL Files and pick the Add button to locate and select the FirstApplication.DCL file. Then pick the Next button to continue on.



Resource Files to Include

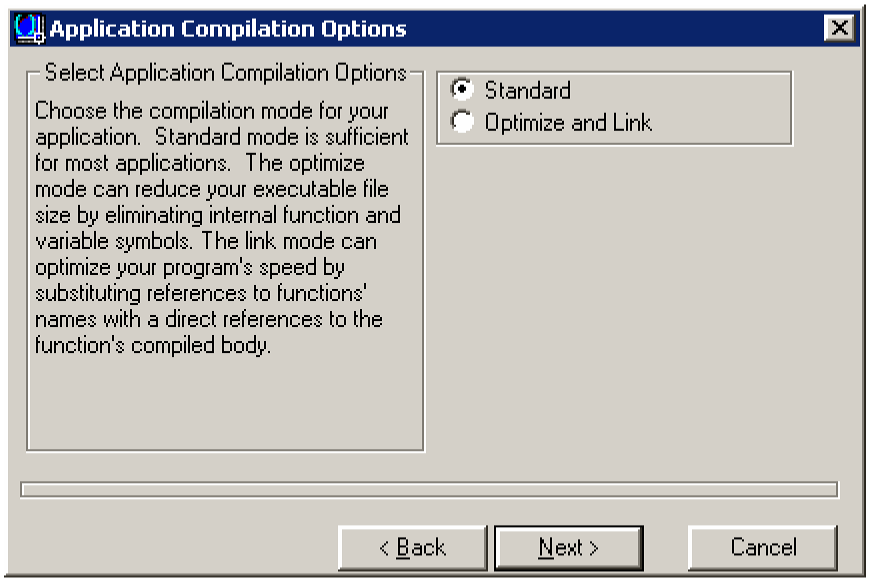
Select Additional Resource Files

Select additional resource files, such as Visual Basic for Application files (.dvb) or Dialog Control Language files (.dcl). These files are auxiliary files for your application and may be loaded from your program.

:DCL C:/.../FirstApplication.dcl

Figure 13-5 – Resource Files Include form

The next panel prompts you to choose either Standard or Optimized compilation. For this example, use the Standard option and pick Next to continue.



Application Compilation Options

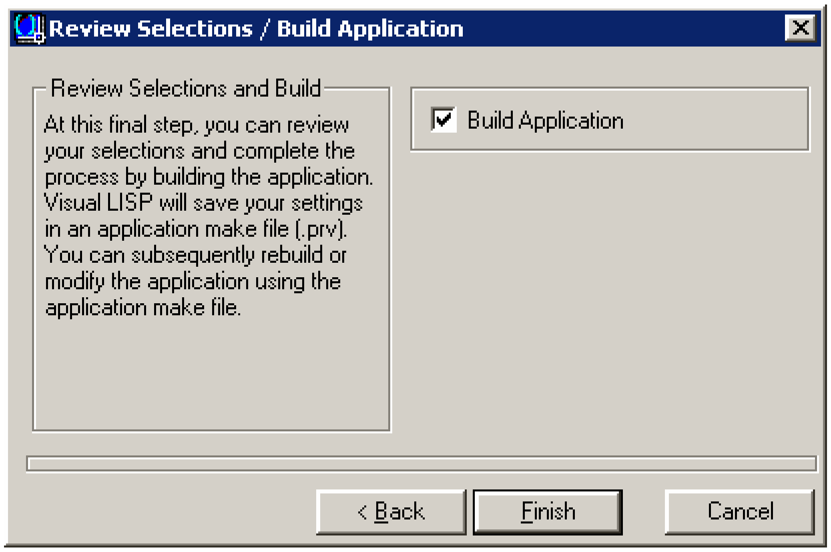
Select Application Compilation Options

Choose the compilation mode for your application. Standard mode is sufficient for most applications. The optimize mode can reduce your executable file size by eliminating internal function and variable symbols. The link mode can optimize your program's speed by substituting references to functions' names with a direct references to the function's compiled body.

● Standard
○ Optimize and Link

Figure 13-6 – Application Compilation Options form

The final form asks if you want to save the Make Application settings and go ahead and compile the VLX application. If you choose not to compile, the settings you just configured are saved to a make file that uses a .PRV file extension. You can reuse make files at any time to recompile using the stored settings and save a lot of time. For this example, go ahead and compile your VLX application by picking the Finish button.


Review Selections / Build Application

Review Selections and Build

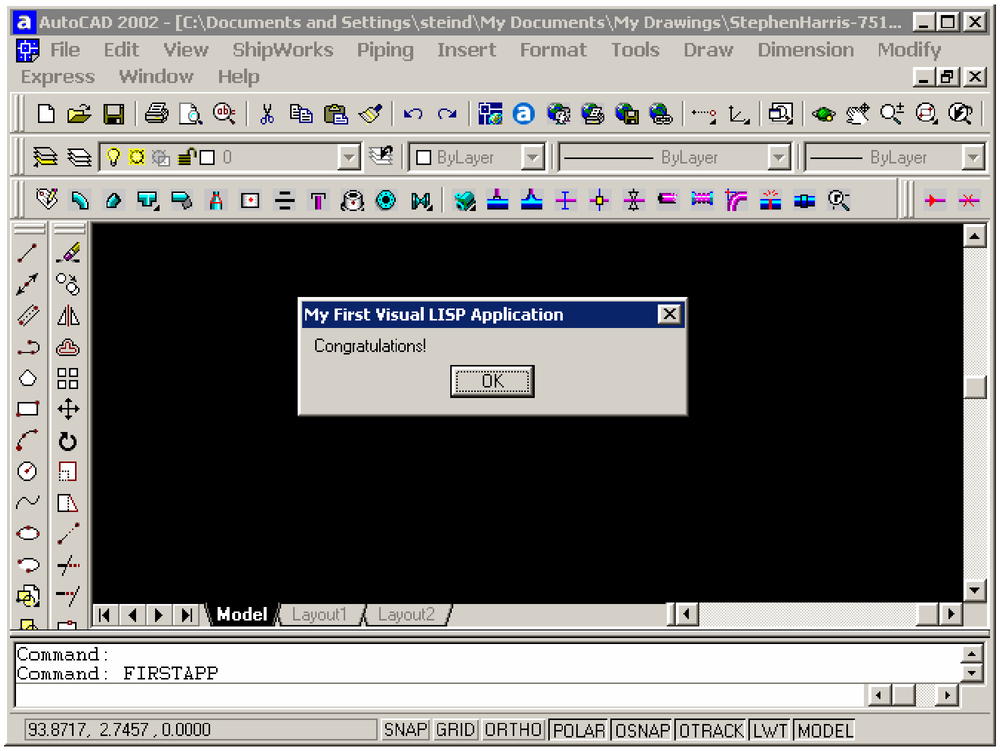
At this final step, you can review your selections and complete the process by building the application. Visual LISP will save your settings in an application make file (.prv). You can subsequently rebuild or modify the application using the application make file.

☑ Build Application

Figure 13-7 – Review selections and build application form.

Now that you've compiled FirstApp.VLX you can load it into AutoCAD and try the FIRSTAPP command to see how it works. You should see a dialog form with one OK button and a message displayed saying "Congratulations!" in the middle.

If this doesn't happen, review this chapter to make sure you followed all steps correctly and compile and load it again. To reload a separate namespace VLX you first have to unload the existing definition by using the (vl-unload-vlx) function. To unload FIRSTAPP, you would use (vl-unload-vlx "firstapp.vlx") at the command prompt.


AutoCAD 2002 - [C:\Documents and Settings\steind\My Documents\My Drawings\StephenHarris-751...

My First Visual LISP Application

Congratulations!

```
Command:
Command:  FIRSTAPP
```

93.8717, 2.7457 , 0.0000 SNAP GRID ORTHO POLAR OSNAP OTRACK LWT MODEL

Figure 13-8 – Results of running the FIRSTAPP command.

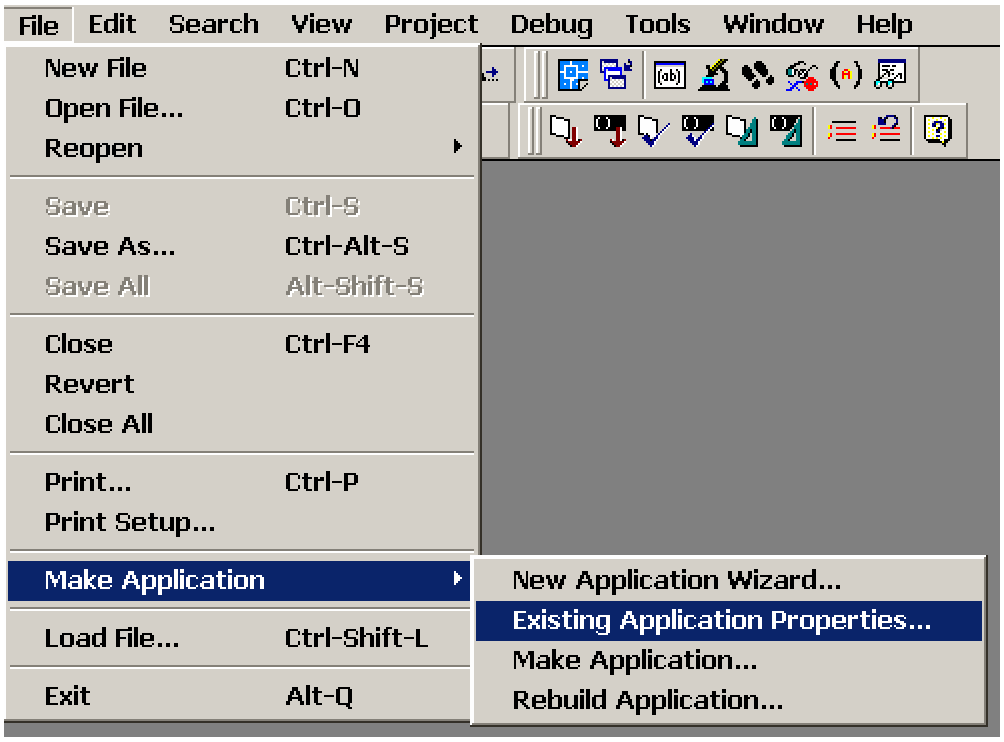
PRV Files

The Make Application Wizard creates a PRV file to store the settings for your application. If you open a PRV file in notepad, you will see that it is actually a LISP formatted file in which the properties are stored as dotted-pair lists. The example below shows a PRV that compiles a LSP and a DCL file into the ASW_PM.VLX output.

```
;;; Visual LISP make file [V1.0] asw_pm saved to:[C:/ASW/SYS]
at:[3/15/02]
(PRV-DEF (:target . "asw_pm.VLX")
        (:active-x . T)
        (:separate-namespace)
        (:protected . T)
        (:load-file-list (:lsp "source/asw_pm.lsp"))
        (:require-file-list (:DCL "source/asw_pm.DCL"))
        (:ob-directory)
        (:tmp-directory)
        (:optimization . st)
)
;; EOF
```

Warning!

Although you might be tempted to “tweak” PRV files in a text editor, you should instead use the “Existing Application Properties” feature to modify the PRV configuration settings. Editing the PRV file manually may corrupt the file and cause errors when you attempt to recompile.



Chapter 14 – Using ObjectDBX with Visual LISP

Visual LISP can interface with any ActiveX-enabled resources available to the user. ObjectDBX is yet another resource provided within AutoCAD that can be tapped by Visual LISP to perform special tasks that are not possible with any other technologies. First, we need to start off by explaining what ObjectDBX really is.

What is ObjectDBX?

ObjectDBX is a subset of ObjectARX, well, sort of. It's a C++ object-oriented API for manipulating AutoCAD and it's related objects, collections, properties, methods and events. While ObjectDBX is capable of many powerful feats of daring, it does have some limitations compared to ObjectARX and Visual LISP. Aside from the limitation, it also provides some nice advantages over them as well. Confused? I know I was at first. But, one place where ObjectDBX really shines is in the world of remote document access, in particular, mining drawings other than those that are open.

Recently, Autodesk released the ObjectDBX SDK for developers to use for mining and manipulating drawing data without having AutoCAD installed. Free? Of course not. In fact, there's a steep price tag and licensing royalty to contend with if you want to pursue this baby. You could opt for OpenDWG alternatives, but since ObjectDBX is built by Autodesk, you can be fairly certain it's going to be reliable when it comes to recognizing all the subtle things in a DWG file.

For the sake of this chapter however, I am going to focus on ObjectDBX as an integral service within AutoCAD, and how it can be used from Visual LISP to perform certain tasks that VLISP alone cannot do. Sound interesting? Let's see how this works.

How to Use ObjectDBX within Visual LISP

In order to use ObjectDBX within Visual LISP, you must first load the ObjectDBX TypeLib interface as shown in Figure 14-1. Then you must invoke the interface using a special function `(vla-getInterfaceObject)` as shown in Figure 14-2. Figure 14-1 shows a few example functions for loading the ObjectDBX TypeLib interface within Visual LISP.

Let's suppose for example, that you would like to be able to search a directory of drawing files to find those that contain a specific block insertion. While you could open each drawing and fetch the Blocks table or do a `(ssget)` search, there is another way to do this without ever opening the drawings in the AutoCAD editor: ObjectDBX.

```
;; Calls REGSVR32 to register a DLL silently via the /S option
(defun DLLRegister (dll) (startapp "regsvr32.exe" (strcat "/s \"" dll "\"")))

;; Calls REGSVR32 to un-register a DLL silently via the /U /S options
(defun DLLUnRegister (dll) (startapp "regsvr32.exe" (strcat "/u /s \"" dll "\"")))

;; Returns the ProgID for a given ClassID if found in registry
(defun ProgID->ClassID (ProgID)
  (vl-registry-read (strcat "HKEY_CLASSES_ROOT\\" progid "\\CLSID"))
)

;; Registers ObjectDBX (if not already), Returns ProgID if successful
(defun DBX-Register ( / classname)
  (setq classname "ObjectDBX.AxDbDocument")
  (cond
    ( (ProgID->ClassID classname) )
    ( (and
        (setq server (findfile "AxDb15.dll"))
        (DLLRegister server)
        (ProgID->ClassID classname)
      )
      (ProgID->ClassID classname)
    )
    ( (not (setq server (findfile "AxDb15.dll")))
      (alert "Error: Cannot locate ObjectDBX Type Library (AxDb15.dll)...")
    )
    ( T
      (DLLRegister "ObjectDBX.AxDbDocument")
      (or
        (ProgID->ClassID "ObjectDBX.AxDbDocument")
        (alert "Error: Failed to register ObjectDBX ActiveX services...")
      )
    )
  )
)
```
Figure 14-1 – Visual LISP functions to load the ObjectDBX interface.

The `(dllregister)` function is a general-purpose tool you can use to perform a Windows DLL registration on a client using the REGSVR32 command through a shell operation. The /S parameter denotes a silent registration which suppresses any notifications during the registration process.

The `(dllunregister)` function performs the opposite of `(dllregister)`, whereby it removes a DLL's registration from a local machine. This is often useful for removing a DLL when you need to register an updated version of the same DLL.

The `(progid->classid)` function performs a look-up of a given class registration in the Windows Registry and returns the GUID, which is a lengthy encoded unique identifier for a given ActiveX component. No two GUID values are the same as they are generated by a complex hashing algorithm during compilation. This particular function verifies that a given DLL has been registered by checking for its GUID in the registry. If no GUID is found, the DLL has not been registered yet, and this returns *nil*. Then you can use `(dllregister)` to register the DLL on the client machine.

The following function in Figure 14.1A opens a remote drawing document and returns the DBX document object if successful, otherwise it returns *nil*. You can use this function to take care of the messy stuff and simply use the returned document object to perform any operations you desire.

```
(defun DBX-doc-open (filename / dbxdoc)
  (cond
    ( (findfile filename)
      (if (not (DBX-Register))
        (exit)
      )
      (setq dbxdoc
        (vla-getinterfaceobject
          (vlax-get-acad-object) "ObjectDBX.AxDbDocument"))
      (cond
        ( (vl-catch-all-error-p
            (vl-catch-all-apply
              'vla-Open (list dbxdoc (findfile filename))
            )
          )
          (princ "\nUnable to open drawing.")
          (exit)
        )
        ( T dbxdoc )
      )
    )
  )
)
```

Figure 14.1A – ObjectDBX Document Opening Function

Now you have a nice little black-box function to open drawings remotely, so you can move on to wrapping inside bigger and better things, like returning table lists and so forth. You can also modify certain properties of remote drawings through DBX.

```
(defun DBX-GetTableList
  (filename tblname / dbxdoc out name)
  (cond
    ( (setq dbxdoc (DBX-doc-open filename))
      (vlax-For tblItem (DBX-TableGet tblName dbxdoc)
        (setq name (vla-get-Name tblItem))
        (if (/= (substr name 1 1) "*")
          (setq out (cons name out))
        )
      )
      (vlax-release-object dbxdoc)
    )
    ( T
      (strcat (princ "\nUnable to open file: " filename))
    )
  )
  (if out (reverse out))
)

(defun DBX-TableGet (tName object)
  (cond
    ( (= (strcase tName) "BLOCKS")     (vla-get-Blocks     object) )
    ( (= (strcase tName) "LAYERS")     (vla-get-Layers     object) )
    ( (= (strcase tName) "TEXTSTYLES") (vla-get-textstyles object) )
    ( (= (strcase tName) "DIMSTYLES")  (vla-get-dimstyles  object) )
    ( (= (strcase tName) "LINETYPES")  (vla-get-linetypes  object) )
    ( (or
       (= (strcase tName) "PLOTCONFIGURATIONS")
       (= (strcase tName) "PAGESETUPS")
      )
      (vla-get-plotconfigurations object)
    )
    ( (= (strcase tName) "LAYOUTS") (vla-get-Layouts object) )
    ( (= (strcase tName) "GROUPS")  (vla-get-Groups  object) )
    ( T
      (vl-exit-with-error "\n(dbx-dwgscan error): Invalid table name specified.")
    )
  )
)
```

Figure 14-2 – Visual LISP functions for using ObjectDBX to inspect drawing tables.

The functions shown in Figure 14-2 use the ObjectDBX “Open” method to access a given drawing file and access a given table collection within it. Among the limitations of using ObjectDBX is that you cannot access tables within any drawings you have opened in your AutoCAD Documents collection, as this will generate an error. ObjectDBX enables access even if a drawing is opened by another user, as long as it is not opened by the user that is requesting to open the drawing through an ObjectDBX interface.

```
(defun DWGSCAN
  ($table $name $dwgfiles / $files $dwgs $path $collection n out)
  (cond
    ( (and $table $name $dwgfiles)
      (princ
        (strcat
          "\nScanning "
          (itoa (length $dwgfiles))
          " drawings for "
          (strcase (substr $table 1 (1- (strlen $table))) t)
          " [" $name "]..."
        )
      )
      (foreach n $dwgfiles
        (cond
          ( (setq $collection (DBX-GetTableList n $table))
            (cond
              ( (member (strcase $name) (mapcar 'strcase $collection))
                (setq out (cons n out))
              )
            )
            (setq $collection nil)
          )
          ( T (princ "\nUnable to query table collection in target drawing.") )
        )
      )
    )
    ( T (princ "\nUsage: (DWGSCAN tablename itemname drawingfiles)"))
  )
  (if out (reverse out))
)
```

Figure 14-3 - Sample Visual LISP function using ObjectDBX methods

Figure 14-3 shows a function that uses the functions in Figures 14-1 and 14-2 to perform a search of a list of drawings for a specified table item. If you load the example file dbx-dwgscan.lsp into your AutoCAD session, you can use the (dwgscan) function to search for items in other drawings. The example below demonstrates using (dwgscan) to search a list of drawings for a block named “Chair123”.

```
Command: (dwgscan "Blocks" "Chair123" dwgfiles)

Scanning 51 drawings for block [Chair123]...
("c:\\drawings\\plan003.DWG"
"c:\\drawings\\plan004.DWG"
"c:\\drawings\\plan005.DWG")
```

Some things to note about using ObjectDBX services from Visual LISP:

- You cannot perform selection set operations on drawings through DBX. Only table operations can be used.
- You cannot open any documents that are opened in the Documents collection of the AutoCAD session performing the DBX operation.
- ObjectDBX does not support using any “command” operations on documents.

- Be sure to release a DBX object when finished using it, and use `(gc)` following any object release of an external process (external to the AutoCAD namespace).

The document interface exposed through ObjectDBX is quite a bit more restrictive than that of a document object internal to a given AutoCAD editing session. Below is a table of the exposed properties and methods of an ObjectDBX Document object. You can see this yourself by performing (vlax-dump-object) on an active DBX document object, such as that returned by the (DBX-Doc-Open) function shown above.

Some interesting notes: Document-centric system variables are not exposed. The Application object is also not present as the document is not actually opened in the Application namespace in a manner like a drawing opened for editing. The Name property is NOT read-only. Notice the methods that are available.

```
; IAxDbDocument: IAxDbDocument Interface
; Property values:
;   Application (RO) = Exception occurred
;   Blocks (RO) = #<VLA-OBJECT IAcadBlocks 037aad64>
;   Database (RO) = #<VLA-OBJECT IAcadDatabase 037ac634>
;   Dictionaries (RO) = #<VLA-OBJECT IAcadDictionaries 037a8a34>
;   DimStyles (RO) = #<VLA-OBJECT IAcadDimStyles 037a8954>
;   ElevationModelSpace = 0.0
;   ElevationPaperSpace = 0.0
;   Groups (RO) = #<VLA-OBJECT IAcadGroups 037acd24>
;   Layers (RO) = #<VLA-OBJECT IAcadLayers 037acc44>
;   Layouts (RO) = #<VLA-OBJECT IAcadLayouts 037acba4>
;   Limits = (0.0 0.0 12.0 9.0)
;   Linetypes (RO) = #<VLA-OBJECT IAcadLineTypes 037a8e84>
;   ModelSpace (RO) = #<VLA-OBJECT IAcadModelSpace 037a8dd4>
;   Name = "C:\\Documents and Settings\\steind\\My Documents\\DRAWING3.dwg"
;   PaperSpace (RO) = #<VLA-OBJECT IAcadPaperSpace 037a8d24>
;   PlotConfigurations (RO) = #<VLA-OBJECT IAcadPlotConfigurations 037a8bf4>
;   Preferences (RO) = #<VLA-OBJECT IAcadDatabasePreferences 037ac694>
;   RegisteredApplications (RO) = #<VLA-OBJECT IAcadRegisteredApplications 037a8b34>
;   TextStyles (RO) = #<VLA-OBJECT IAcadTextStyles 037a93a4>
;   UserCoordinateSystems (RO) = #<VLA-OBJECT IAcadUCSs 037a92f4>
;   Viewports (RO) = #<VLA-OBJECT IAcadViewports 037a91e4>
;   Views (RO) = #<VLA-OBJECT IAcadViews 037a9124>
; Methods supported:
;   CopyObjects (3)
;   DxfIn (2)
;   DxfOut (3)
;   HandleToObject (1)
;   ObjectIdToObject (1)
;   Open (1)
;   RegisterLicenseObject (2)
;   RevokeLicenseObject (1)
;   Save ()
;   SaveAs (1)
```

Chapter 15 – XDATA and XRECORDs

AutoCAD provides a few ways to store information in drawings that is non-graphical. This can include a variety of data types such as numbers, text and so forth. Of these, the two most common are Extended Entity Data (EED) and Dictionaries. The most common form of EED is XDATA, which is an extension of all graphical entities as well as many table objects such as Layers and Linetypes. This allows you to store hidden (non-graphical) information within these entities or table objects and retrieve the information when required.

Another form of storing non-graphical information is through the use of XRECORD objects. XRECORD objects are part of the Document object and allow you to store string information within a Dictionary collection.

The advantages of using XDATA are that the information is attached to a specific entity or table member. The advantages to using XRECORD objects is that they are attached to the Document itself, and not to any particular entity or table object. In addition, XDATA has certain limits on the size of the data that can be stored on a given entity or table member. XRECORD objects do not impose any size limitation on data storage, but it does affect the DWG file size, and memory requirements when the drawing is opened.

Working with XDATA

Xdata can be attached to and retrieved from any graphical entity in a drawing, as well as many table objects, such as layers, layouts and linetypes. Xdata divides information storage by the data type, so you have to be aware of the type of information you intend to store whenever you attach it to anything, as well as when you attempt to retrieve it. For example, if you attach an integer value to an entity and attempt to retrieve it as though it were a string value, you will not get the desired results.

Working with XRECORD Objects

XRECORD objects are maintained as a Dictionary, meaning they have a unique name and are accessed by that name. They are attached to the document object itself, not to any graphic objects or tables as is the case with Xdata. Xrecords are stripped from a drawing if it is saved back to R12 or often when converted to another CAD format that does not support them.

Because XRECORD objects are attached to the document, they are safe from casual deletion by users. For example, if you attach XDATA to a layer, and that layer is purged, the XDATA is then released as well. You can attach XDATA to layer “0” to prevent this, however, XDATA still imposes limits on data type and data size that can be stored.

Xrecords can only be created, renamed or deleted. There are no direct methods for modifying them. The only way to modify an Xrecord is to retrieve its contents, delete the Xrecord from the dictionary object, and recreate a new Xrecord with new data. The following functions demonstrate how to do this using standard AutoLISP.

```
(defun Xrecord-Rebuild (name dat)
  (Xrecord-Delete name)
  (Xrecord-Add name dat)
)

(defun Xrecord-Get (name / xlist)
  (if (setq xlist (dictsearch (namedobjdict) name))
    (cdr (assoc 1 xlist))
  )
)

(defun Xrecord-Delete (name)
  (dictremove (namedobjdict) name); remove from dictionary
)

(defun Xrecord-Add
  (name sdata / xrec xname)
  (setq xrec
    (list
      (cons 0 "XRECORD")
      (cons 100 "AcDbXrecord")
      (cons 1 sdata)
      (cons 62 1)
    )
  )
  (setq xname (entmakex xrec)); rebuild xrecord
  (dictadd (namedobjdict) name xname); return to dictionary
  (princ)
)
```

The problem with the above form is that it uses (entmake) and can sometimes cause problems in AutoCAD when mixed with certain other ActiveX functions. A more appropriate form would be the ActiveX approach as shown in the examples below.

```
(vl-load-com)

(defun Xrecord-Rebuild (dict name data)
  (Xrecord-Delete dict name)
  (Xrecord-Add dict name data)
)

(defun Xrecord-Get
  (dict name / acad doc dcs odc xrec #typ #dat out)
  (setq acad  (vlax-get-acad-object)
        doc   (vla-get-activedocument acad)
        dcs   (vla-get-dictionaries doc)
  )
  (cond
    ( (setq odc (dsx-item dcs dict))
      (cond
        ( (setq xrec (dsx-item odc name))
          (vla-getXrecordData xrec '#typ '#dat)
          (setq #typ (vlax-safearray->list #typ)
                #dat (vlax-safearray->list #dat)
          )
          (setq out (mapcar 'vlax-variant-value #dat))
         (vlax-release-object odc)
        )
      )
      (vlax-release-object dcs)
    )
  )
  out
)

(defun Xrecord-Delete (dict name / dcs odc xr)
  (setq dcs (vla-get-dictionaries active-doc))
  (cond
    ( (setq odc (dsx-item dcs dict))
      (cond
        ( (setq xr (dsx-item odc name))
          (vla-delete xr)
          (vlax-release-object xr)
        )
      )
      (vlax-release-object odc)
    )
  )
  (vlax-release-object dcs)
)

(defun Xrecord-Add
  (dict name data / acad doc dicts dict xrec #typ #dat)
  (setq acad  (vlax-get-acad-object)
        doc   (vla-get-activedocument acad)
        dicts (vla-get-Dictionaries doc)
        dict  (vlax-invoke-method dicts "Add" dict)
        xrec  (vla-AddXrecord dict name)
  )
  (if (not (listp data)) (setq data (list data))); ensure list!
  (vla-setXrecordData xrec
    (List->VariantArray
      (List->IntList data)
      'vlax-vbInteger
    )
    (List->VariantArray data 'vlax-vbVariant)
  )

  (vla-getXrecordData xrec '#typ '#dat)
  (setq #typ (vlax-safearray->list #typ)
        #dat (vlax-safearray->list #dat)
  )
  (mapcar 'vlax-variant-value #dat)
)
```

The two functions `(List->VariantArray)` and `(List->IntList)` are used to define the safearray contents and dimension respectively. They can be used for much more than this obviously. The second argument to `(List->VariantArray)` must be a single-quoted ActiveX data type declaration such as `‘vlax-vbString`.

```
(defun List->VariantArray (lst datatype / arraySpace sArray)
  (setq arraySpace
         (vlax-make-safearray
           (eval datatype)
           (cons 0 (1- (length lst)))
         )
  )
  (setq sArray (vlax-safearray-fill arraySpace lst))
  (vlax-make-variant sArray)
)

(defun List->IntList (lst / n)
  (setq n 0)
  (mapcar (function (lambda (x) (setq n (1+ n)))) lst)
)
```

The other function `(dsx-item)` is used to fetch (or attempt to fetch) an item using the Item method of a collection. This function includes error catching in case the fetch fails, which returns an ActiveX error instead of something like *nil*. In this case, we trap an error and return *nil* if the fetch fails. Otherwise, we return the object from the collection.

```
(defun DSX-Item (collection item / out)
  (if
    (not
      (vl-catch-all-error-p
        (setq out
          (vl-catch-all-apply 'vla-item (list collection item))
        )
      )
    )
    out ; return object or nil
  )
)
```

To demonstrate how to use this stuff we'll save an Xrecord in our current drawing with some information such as the current username (assuming we're on Windows NT, 2000 or XP) and some other information.

```
(setq username (getenv “username”); logged on user ID
      machine  (getenv “computername”); NETBIOS computer name
)
(Xrecord-Rebuild “PERSONAL” “UserData” (list username machine))
_$ Returns (“DSTEIN1234” “W2K-1234”)

(Xrecord-Get “PERSONAL” “UserData”)
_$ Returns (“DSTEIN1234” “W2K-1234”)

(Xrecord-Rebuild “PERSONAL” “UserData” “1234”)
_$ Returns (“1234”)
```

So, what can you do with Xrecords? Anything you want. They are very useful for storing information in a drawing that is not tied directly to any particular entity or table. If you are used to storing information in Xdata, you are probably aware that if the entity or table item is deleted the Xdata is lost. Of course, you can attach Xdata to things like Layer “0” or the like, so it never gets deleted. However, Xdata imposes limitations on contents that might be alleviated by switching to Xrecords instead.

Chapter 16– The AutoCAD Application Object

The AcadApplication object is the root of everything Visual LISP can address relating to ActiveX. This includes collections, properties, methods, events and derived objects that inherit from this base object. Remember, the ActiveX object model is a hierarchy of classes that allow for lower level classes to be derived that contain properties and methods of their parent classes. In order to gain access to anything within AutoCAD, you must first start by gaining access to AutoCAD itself, and then work your way down into whatever you need.

For example, if you want to access the Layers collection and fetch a particular layer, you must first get the AcadApplication object, then the ActiveDocument object, and then get the Layers collection from that object. One way to begin understanding the AcadApplication object is by dumping the object using the `(vlax-dump-object)` function. This function takes one required argument (the object) to request a list of its properties, and an optional flag, which if provided (and non-nil) requests a list of methods as well.

```
Command: (vlax-dump-object (vlax-get-acad-object) T)
; IAcadApplication: An instance of the AutoCAD application
; Property values:
;   ActiveDocument = #<VLA-OBJECT IAcadDocument 00ed7e0c>
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;   Caption (RO) = "AutoCAD 2002 - [C:\\Documents and Settings\\...\\Drawing1.dwg]"
;   Documents (RO) = #<VLA-OBJECT IAcadDocuments 00f20ef0>
;   FullName (RO) = "C:\\Program Files\\AutoCAD 2002\\acad.exe"
;   Height = 723
;   LocaleId (RO) = 1033
;   MenuBar (RO) = #<VLA-OBJECT IAcadMenuBar 053247f4>
;   MenuGroups (RO) = #<VLA-OBJECT IAcadMenuGroups 01433208>
;   Name (RO) = "AutoCAD"
;   Path (RO) = "C:\\Program Files\\AutoCAD 2002"
;   Preferences (RO) = #<VLA-OBJECT IAcadPreferences 053277fc>
;   StatusId (RO) = ...Indexed contents not shown...
;   VBE (RO) = #<VLA-OBJECT VBE 03d618f4>
;   Version (RO) = "15.06"
;   Visible = -1
;   Width = 1032
;   WindowLeft = -4
;   WindowState = 3
;   WindowTop = -4
; Methods supported:
;   Eval (1)
;   GetAcadState ()
;   GetInterfaceObject (1)
;   ListArx ()
;   LoadArx (1)
;   LoadDVB (1)
;   Quit ()
;   RunMacro (1)
;   UnloadArx (1)
;   UnloadDVB (1)
;   Update ()
;   ZoomAll ()
;   ZoomCenter (2)
;   ZoomExtents ()
;   ZoomPickWindow ()
;   ZoomPrevious ()
;   ZoomScaled (2)
;   ZoomWindow (2)
```

Figure 16-1 – (vlax-dump-object) results on the AcadApplication object.

While you might expect the list of Properties, Collections and Methods to be much larger for the AcadApplication object, remember that the Object Model is a tree structure. This means that much of the complexity is delegated at multiple levels, such as down into the Documents collection, the Preferences collection and so forth. The items shown in Figure 16-1 apply *only* to the AcadApplication object and nothing else.

To demonstrate a tiny bit of what you can do with the AcadApplication object, load and run the following sample of code. This will minimize the AutoCAD session window and then maximize it after a short pause.

```
(defun MinMax ( / acad)
  (vl-load-com)
  (setq acad (vlax-get-acad-object))
  (vla-put-windowstate acad acMin)
  (vl-cmdf “DELAY” 1000)
  (vla-put-windowstate acad acMax)
  (vlax-release-object acad)
)
```

Minimizing AutoCAD can come in handy when you intend to write a program that launches another application. Quite often, AutoCAD will jump back to the front and hide the other application window because it attempts to regain “focus” from the Windows application stack. This doesn’t always happen, but it happens frequently. One way to avoid this is to hide AutoCAD after you launch the other application. This will prevent it from popping back in front of the other application window.

```
(defun ShowNotepad (filename / acad fn)
  (vl-load-com)
  (cond
    ( (setq fn (findfile filename)); make sure file exists first
      (setq acad (vlax-get-acad-object))
      (vla-put-windowstate acad acMin)
      (vlax-release-object acad)
      (startapp “notepad.exe” fn)
    )
    ( T (princ (strcat “\nFile not found: “ filename)) )
  )
)
```

Another solution is to use a third-party function such as the DOSlib 6 function `(dos_exewait)` which launches another application and suspends AutoCAD until the other application session is terminated (closed).

Tip! The Path property of the AcadApplication object shows the path to where ACAD.EXE resides on the local machine. This can be used to get the actual installation path when performing modifications to the support files path list within the AcadPreferences *Files* collection.

Here’s a twist. Let’s say you have a LISP or VLX program that needs to interface with a VBA application. The VBA application is loaded as a DVB project macro or macro collection. Maybe you’d like to be able to query AutoCAD to see if the DVB is currently loaded and available for use. How can this be done? Easy. If you look at the properties list above, you’ll see one named VBE. What’s this? VBE is the Visual Basic Environment object. You can access this object and request a list of loaded projects from it using VLISP:

```
(defun VBA-Loaded ( / vb vbp vbc out vbe i vb1 vbn)
  (setq vb  (vla-get-vbe (vlax-get-acad-object))
        vbp (vlax-get-property vb “vbprojects”)
  )
  (if (> (vla-get-count vbp) 0)
    (progn
      (setq i 1)
      (repeat (vla-get-count vbp)
        (setq vbe (vla-item vbp i)
              vbn (vlax-get-property vbe "filename")
              vb1 (strcat
                    (vl-filename-base vbn)
                    (vl-filename-extension vbn)
                  )
              out (cons vb1 out)
              i   (1+ i)
        )
        (vlax-release-object vbe)
      )
    )
  )
  (vlax-release-object vb)
  (if out (acad_strlsort out))
)
```

Note that you can only access the “vbprojects” property using `(vlax-get-property)`. For some reason, this is one of the few properties within AutoCAD that won’t work with `(vla-get-property)`. This function will return as list of loaded DVB projects in the following manner:

```
_$ (VBA-Loaded) Returns: ("name1.dvb" "name2.dvb"...)
```

If you inspect the VBE object you’ll find some interesting properties…

```
(vlax-dump-object (vla-get-vbe (vlax-get-acad-object)) t)
; VBE: nil
; Property values:
;   ActiveCodePane = nil
;   ActiveVBProject = nil
;   ActiveWindow (RO) = nil
;   Addins (RO) = #<VLA-OBJECT _AddIns 046a1a44>
;   CodePanes (RO) = #<VLA-OBJECT _CodePanes 001846f8>
;   CommandBars (RO) = #<VLA-OBJECT _CommandBars 0510a9f0>
;   Events (RO) = #<VLA-OBJECT Events 046a1984>
;   MainWindow (RO) = #<VLA-OBJECT Window 046a1b84>
;   SelectedVBComponent (RO) = nil
;   VBProjects (RO) = #<VLA-OBJECT _VBProjects 046a1934>
;   Version (RO) = "6.03"
;   Windows (RO) = #<VLA-OBJECT _Windows 046a1a04>
; No methods
```

Tip! Get the `ActiveCodePane` object and drill down into the `CodeModule` object and continue on from there. Very powerful stuff if you find a way to put it to use.

Chapter 17 – AutoCAD Entities

AutoCAD Entities are graphical objects such as ARC, CIRCLE and LINE objects. The term entity is synonymous with graphical object in the context of ActiveX programming within AutoCAD. All entities are derived from the base class named AcDbEntity, which provides certain default properties and methods to all entitles. Some default properties are Layer and Color, while some default methods are Copy and Delete.

Some properties are common to all objects. Some properties are common to groups of objects, but not all objects. Some properties are specific to a given object type. To access or modify object properties using Visual LISP, use the following examples:

(vla-get-*propertyname* object) where propertyname might be color or linetype. You can also use the alternate form **(vlax-get-*property* object *propertyname*)** if desired. These are interchangeable when it comes to working with AutoCAD objects.

(vla-put-*propertyname* object *value*) where propertyname might be color or linetype and value might be acRed or "dashed". You can also use the alternate form **(vlax-put-property object *propertyname value*)** if desired. Again, these are interchangeable when working with AutoCAD objects.

All Objects – Common Properties

Property	**Description**	**Data Type**	**DXF**
Application (RO)	AutoCAD session	Object (pointer)	
Color	Entity color	Integer/Enum	62 (transient)
Document (RO)	Parent document	Object (pointer)	
Handle	Entity Handle ID	String	5
HasExtensionDictionary (RO)	Has XDATA attached	Boolean	
Hyperlinks (RO)	Web links	Object (collection)	
Layer	Entity layer	String	8
Linetype	Entity linetype	String	6 (transient)
LinetypeScale	Entity linetype scale	Real/Double	
Lineweight	Entity lineweight	Integer/Enum	
ObjectID (RO)	Object ID value	Integer	
ObjectName (RO)	Name of Entity type	String	0
OwnerID (RO)	Parent object	Integer	
Normal	Extrusion Vector	Array	210
PlotStyleName	Name of PlotStyle	String	
Thickness	Entity thickness	Real/Double	39 (transient)
Visible	Toggles display	Boolean	
Layout-Name		(N/A)	410

** Depends upon use of Color-Based or Named plotstyles in active drawing.

The ARC object

Property	**Description**	**Data Type**	**DXF**
ArcLength (RO)	Perimeter length	Double	
Area	Enclosed area	Double	
Center	Center Point	Array	10
EndAngle	Ending Angle	Double	52
EndPoint	End Point	Array	
Radius	Radius value	Double	40
StartAngle	Starting Angle	Double	51
StartPoint	Start point	Array	
TotalAngle (RO)	Total angle in radians	Double	

Special notes about ARC objects:

You cannot change the start point of an arc or ellipse. To edit an arc, use the EndAngle and Radius properties. To edit an ellipse, use the EndAngle, MajorAxis, and RadiusRatio properties.

The Area property is calculated in square drawing units as though it were closed by a vector from the startpoint to the endpoint. This is not the same as the area of a circular segment, which would include the area from the centerpoint out to the perimeter as a portion of the total circular area.

The CIRCLE entity

Property	Description	Data Type	DXF
Area	Enclosed area	Double	
Center	Center Point	Array	10
Diameter	Diameter value	Double	
Radius	Radius value	Double	40

The RotatedDimension (LinearDimension) entity

Property	Description	Data Type	DXF
AltRoundDistance		Double	
AltSuppressLeadingZeros		Integer	
AltSuppressTrailingZeros		Integer	
AltSuppressZeroFeet		Integer	
AltSuppressZeroInches		Integer	
AltTextPrefix		String	
AltTextSuffix		String	
AltTolerancePrecision		Integer	
AltToleranceSuppressLeadingZeroes		Integer	
AltToleranceSuppressTrailingZeroes		Integer	
AltUnits		Integer	
AltUnitsFormat		Integer	
AltUnitsPrecision		Integer	
AltUnitsScale		Double	
Arrowhead1Block		String	
Arrowhead1Type		Integer	
Arrowhead2Block		String	
Arrowhead2Type		Integer	
ArrowheadSize		Double	
DecimalSeparator		String	
DimensionLineColor		Integer	
DimensionLineExtend		Double	
DimensionLineWeight		Integer/Enum	
DimLine1Suppress		Integer/Boolean	
DimLine2Suppress		Integer/Boolean	
DimLineInside		Integer	
ExtensionLineColor		Integer/Enum	
ExtensionLineExtend		Double	
ExtensionLineOffset		Double	
ExtensionLineWeight		Integer/Enum	
ExtLine1Suppress		Integer/Boolean	
ExtLine2Suppress		Integer/Boolean	
Fit		Integer	
ForceLineInside		Integer	
FractionFormat		Integer/Enum	
HorizontalTextPosition		Integer/Enum	
LinearScaleFactor		Double	
Measurement (RO)		Double	
PrimaryUnitsPrecision		Integer	
Rotation		Double	
RoundDistance		Double	

ScaleFactor		Double
StyleName	DimStyle name	String
SuppressLeadingZeroes		Integer
SuppressTrailingZeroes		Integer
SuppressZeroFeet		Integer
SuppressZeroInches		Integer
TextColor	Text color	Integer/Enum
TextGap	Gap distance	Double
TextHeight	Text height	Double
TextInside		Integer
TextInsideAlign		Integer/Enum
TextMovement	Text movement	Integer
TextOutsideAlign		Integer
TextOverride	Override string	String
TextPosition	Text position	Array
TextPrefix		String
TextRotation		Double
TextStyle		String
TextSuffix		String
ToleranceDisplay		Integer
ToleranceHeightScale		Double
ToleranceJustification		Integer/Enum
ToleranceLowerLimit		Double
TolerancePrecision		Integer
ToleranceSuppressLeadingZeroes		Integer/Boolean
ToleranceSuppressTrailingZeroes		Integer/Boolean
ToleranceSuppressZeroFeet		Integer/Boolean
ToleranceSuppressZeroInches		Integer/Boolean
UnitsFormat		Integer/Enum
VerticalTextPosition		Integer/Enum

Notes Regarding Rotated Dimension Objects:

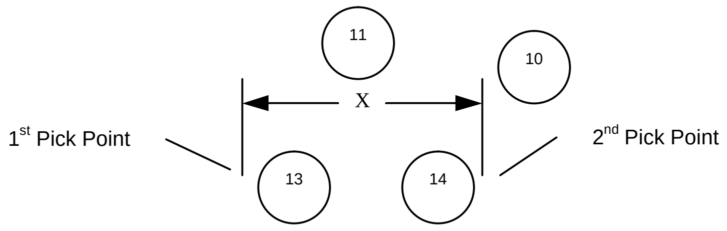
Control points for extension lines and dimension lines are not exposed through ActiveX. To obtain these control points you must use the DXF entity codes 10, 11, 13, and 14. They are described as follows:

13 = Start point of first extension line (first control point)

14 = Start point of second extension line (second control point)

11 = Middle-Center point of MTEXT label

10 = Arrowhead point of second dimension line



The ELLIPSE entity

Property	Description	Data Type	DXF
Area	Enclosed area	Double	
Center	Center Point	Array	10
Diameter	Diameter value	Double	
EndAngle	End Angle (Arc)	Double	
EndParameter	End Parameter		
EndPoint (RO)	Endpoint (Arc)	Array	

MajorAxis	Axis Endpoint	Array	
MajorRadius	Major Radius	Double	
MinorAxis	Axis Endpoint	Array	
MinorRadius	Minor Radius	Double	
RadiusRatio	Ratio of Major/Minor	Double	
StartAngle	Start Angle (Arc)	Double	
StartParameter	Start parameter		
StartPoint (RO)	Start Point (Arc)	Array	

Notes regarding ELLIPSE objects:

If the Ellipse is closed, the StartAngle value is 0 and the EndAngle value is 2*Pi.

You cannot change the Startpoint or Endpoint properties of an Ellipse.

The LEADER entity

Property	**Description**	**Data Type**	**DXF**
StartPoint	Start Point	Array	

The LINE entity

Property	**Description**	**Data Type**	**DXF**
Angle (RO)	Angle	Double	
Delta (RO)	Offset X and Y	Array	
EndPoint	End Point	Array	11
Length (RO)	Length	Double	
StartPoint	Start Point	Array	10

The LWPOLYLINE entity

Property	**Description**	**Data Type**	**DXF**
Area	Enclosed Area	Double	
Closed	Closed flag	Integer/Boolean	
ConstantWidth	Default width	Double	40
Coordinate		Array	
Coordinates	Vertices List	Array/Doubles	10
Elevation	Z-elevation	Double	
LinetypeGeneration	Linetype Gen flag	Integer/Boolean	
Type	Curve Fit Type	Integer/Enum	

Notes regarding LwPolyline entities:

Breaking a LwPolyline results in the remaining pieces being converted into PolyLine entities. The TYPE property is transient, meaning that if the entity has not been curve-fitted (spline, bezier, cubic) this property is not exposed. Once a curve-fitting is applied, the TYPE property is available.

The MLINE entity

Property	**Description**	**Data Type**	**DXF**
Angle (RO)	Angle	Double	
Coordinates	Vertices List	Array/Doubles	10
Delta (RO)	Offset X and Y	Array	
EndPoint	End Point	Array	
Length (RO)	Length	Double	
StartPoint	Start Point	Array	
StyleName	Name of MLINE style	String	

The MTEXT entity

Property	**Description**	**Data Type**	**DXF**

AttachmentPoint	Basepoint	Array	10
DrawingDirection	Text Flow	Integer/Enum	
LineSpacingFactor	Row Spacing Factor	Double	
LineSpacingStyle	Mode to apply factor	Integer/Enum	
Rotation	Rotation Angle	Double	50
StyleName	Style Name	String	7
TextString	String Value	String	1
Width	Frame Width	Double	

Notes Regarding MTEXT objects:

Linespacing is factored using the base Height value. When varying heights are used in a given string, the LineSpacingFactor property is applied against the base Height property value only.

Fraction stacking behavior is controlled by the TSTACKSIZE and TSTACKALIGN system variables. TSTACKSIZE is an integer that denotes "per 100" or percentage of text height the size of the fraction is. A value of 70 denotes (0.7 * Height). TSTACKALIGN controls how stacking is applied. 0=no stacking, 1=diagonal (e.g. ¼), 2=vertical (e.g. horizontal bar).

Formatting Control Codes:

"\\P" denotes line-feed/carriage return

"\\S" denotes fraction stack begin grouping

"\\A" denotes relative height change (using relative factor against base Height property)

The POINT entity

Property	Description	Data Type	DXF
Coordinates	Basepoint	Array	10

Notes Regarding POINT Entities:

The display of POINT entities is controlled by the PDMODE system variable. A setting of 1 hides them. A setting of 0 or another positive number displays them with various symbol types. The PDSIZE system variable controls the relative size of POINT symbols with respect to the zoom factor.

The POLYLINE entity

Property	Description	Data Type	DXF
Area	Enclosed Area	Double	
Closed	Closed flag	Integer/Boolean	
ConstantWidth	Default width	Double	40
Coordinate		Array	
Coordinates	Vertices List	Array/Doubles	10
Elevation	Z-elevation	Double	
LinetypeGeneration	Linetype Gen flag	Integer/Boolean	
Type	Curve Fit Type	Integer/Enum	

The RAY entity

Property	Description	Data Type	DXF
BasePoint	Base point	Array/Doubles	

DirectionVector	Vector Axis Point	Array/Doubles	
SecondPoint	Pick Point	Array/Doubles	

The SOLID entity

Property	Description	Data Type	DXF
Coordinate	?	Array	
Coordinates	Verticies List	Array/Doubles	

The SPLINE entity

Property	Description	Data Type	DXF
Coordinates	Basepoint	Array	

The TEXT entity

Property	Description	Data Type	DXF
Alignment	Alignment Type	Integer/Enum	
Backward	Backwards flag	Integer/Boolean	
Height	Text Height	Double	40
InsertionPoint	Base Point	Array/Doubles	10
ObliqueAngle	Oblique angle	Double	
Rotation	Rotation Angle	Double	50
ScaleFactor	Width Factor	Double	41
StyleName	Text Style Name	String	7
TextAlignmentPoint	Alignment Point	Array/Doubles	11
TextGenerationFlag	Generation Flag	Integer/Enum	
TextString	String Value	String	1
UpsideDown		Integer/Boolean	

The TRACE entity

Property	Description	Data Type	DXF
Coordinate			
Coordinates	Verticies List	Array/Doubles	

The VIEWPORT entity

Property	Description	Data Type	DXF
ArcSmoothness	View Resolution	Integer	
Center	Center Point	Array/Doubles	
Clipped (RO)	Clipped Boundary	Integer/Boolean	
CustomScale	Scale in Mspace		
Direction	View direction vector	Array/Doubles	
DisplayLocked	Locked zoom/pan	Integer/Boolean	
GridOn	Grid display	Integer/Boolean	
Height	Height in Paperspace	Double	
LensLength	Lens length value	Double	
RemoveHiddenLines	Hide lines	Integer/Boolean	
SnapBasePoint	Snap basepoint	Array/Doubles (2D)	
SnapOn	Display snap	Integer/Boolean	
SnapRotationAngle	Snap rotation angle	Double	
StandardScale	ZoomXP factor	Integer/Enum	
Target	Target vector point	Array/Doubles	
TwistAngle	Twist Angle	Double	
UCSIconAtOrigin	Show UCS origin	Integer/Boolean	
UCSIconOn	Show UCS icon	Integer/Boolean	
UCSPerViewport	UCS per viewport	Integer/Boolean	-1 (:vlax-True)
ViewportOn	Display Viewport	Integer/Boolean	-1 (:vlax-True)
Width	Width in Paperspace	Double	

The XLINE entity

Property	Description	Data Type	DXF
BasePoint	Pick Point 1	Array/Doubles	
DirectionVector	Axis Direction Point	Array/Doubles	
SecondPoint	Pick Point 2	Array/Doubles	

Chapter 18 – Documents

Drawings are considered Documents in a general sense and this is how they are referred to from a programmatic point of view within AutoCAD as well as within most other ActiveX-enabled applications. Document objects are members of the Documents collection within AutoCAD.

To access the current drawing session you can request the ActiveDocument property of the AcadApplication object, without having to go to the Documents collection. However, if you need to access another document or iterate through all opened documents, you will need to access the Documents collection.

The Documents Collection

The Documents collection contains all opened documents in the active AutoCAD session. Each time you create or open another drawing, it is immediately added to this collection. Documents are normally entered into the Documents collection in the order they were opened. To navigate a collection, you can use the `(vlax-for)` function to process all members, or use the `(vla-Item)` method to access an individual member by index location or name if desired.

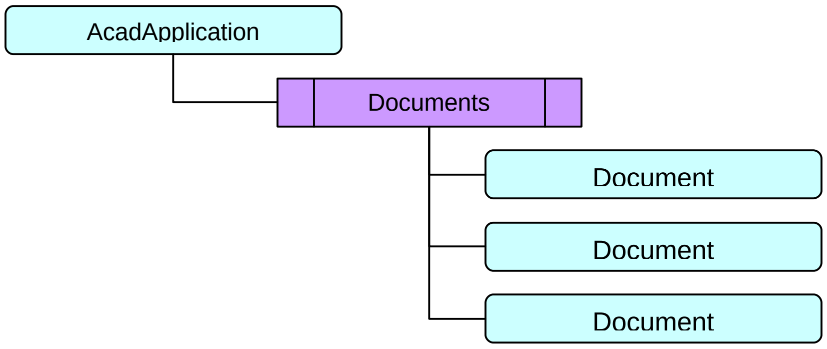


Figure 18-1 – The Documents collection and Document objects.

```
(defun Documents-ListAll ( / out)
  (vlax-for each (vla-get-documents (vlax-get-acad-object))
    (setq out (cons (vla-get-name each) out))
  )
  (if out (reverse out))
)
```
Figure 18-2 – Listing all opened documents by name

Figure 18-2 shows an example code snippet for retrieving a list of all opened document names in the current AutoCAD session. Using the `(vlax-for)` iteration function, which is almost identical to the AutoLISP `(foreach)` function, we can loop through the Documents collection and fetch the Name property of each document and produce a list output.

You could adapt this piece of code very easily to perform other tasks on each document, or to search the documents for a particular condition and act on them as a result.

What else can you do with the Documents collection? Well, let's begin by inspecting what properties and methods the Documents collection supports:

```
_$ (setq docs (vla-get-documents acad))
#<VLA-OBJECT IAcadDocuments 00f20440>

_$ (vlax-dump-object docs T)
; IAcadDocuments: The collection of all AutoCAD drawings open in the
current
session
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;   Count (RO) = 1
; Methods supported:
;   Add (1)
;   Close ()
;   Item (1)
;   Open (2)
```

There are only two properties, but there are four methods. As I mentioned earlier, VLISP does not provide a means for modifying collection properties. It usually sports a set of methods for adding, accessing and deleting members within them however. In this case, the Add method is synonymous with the command NEW, and the Open method is, well, the same as the command OPEN.

To access an individual drawing that you have opened, you can use the Item method with either the name of the document or the index number (its position in the collection). I'll get the object for the "Drawing1.dwg" document that I happen to have opened right now:

```
_$ (setq dwg (vla-item docs "Drawing1.dwg"))
#<VLA-OBJECT IAcadDocument 00ed1e0c>
```

Now I can inspect this document object to see what properties and methods it provides:

```
_$ Command: (vlax-dump-object dwg T)
;IAcadDocument: An AutoCAD drawing
;Property values:
; Active (RO) = -1
; ActiveDimStyle = #<VLA-OBJECT IAcadDimStyle 00f20034>
; ActiveLayer = #<VLA-OBJECT IAcadLayer 00f21c44>
; ActiveLayout = #<VLA-OBJECT IAcadLayout 00f21c04>
; ActiveLinetype = #<VLA-OBJECT IAcadLineType 00f228d4>
; ActivePViewport = AutoCAD: No active viewport in paperspace
; ActiveSelectionSet (RO) = #<VLA-OBJECT IAcadSelectionSet 00f211e4>
; ActiveSpace = 1
; ActiveTextStyle = #<VLA-OBJECT IAcadTextStyle 00f22554>
; ActiveUCS = #<VLA-OBJECT IAcadUCS 00f22234>
; ActiveViewport = #<VLA-OBJECT IAcadViewport 00f22014>
; Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
; Blocks (RO) = #<VLA-OBJECT IAcadBlocks 00f23c04>
; Database (RO) = #<VLA-OBJECT IAcadDatabase 00f202f4>
; Dictionaries (RO) = #<VLA-OBJECT IAcadDictionaries 00f23814>
; DimStyles (RO) = #<VLA-OBJECT IAcadDimStyles 00f23544>
; ElevationModelSpace = 0.0
; ElevationPaperSpace = 0.0
; FullName (RO) = ""
; Groups (RO) = #<VLA-OBJECT IAcadGroups 00f23304>
; Height = 571
; HWND (RO) = 393946
; Layers (RO) = #<VLA-OBJECT IAcadLayers 00f23114>
; Layouts (RO) = #<VLA-OBJECT IAcadLayouts 00f24e94>
; Limits = (0.0 0.0 12.0 9.0)
; Linetypes (RO) = #<VLA-OBJECT IAcadLineTypes 00f24c44>
; ModelSpace (RO) = #<VLA-OBJECT IAcadModelSpace 00f249e4>
; MSpace = AutoCAD: Invalid mode
; Name (RO) = "Drawing1.dwg"
; ObjectSnapMode = 0
; PaperSpace (RO) = #<VLA-OBJECT IAcadPaperSpace 00f244d4>
; Path (RO) = "C:\\Program Files\\AutoCAD 2002"
; PickfirstSelectionSet (RO) = #<VLA-OBJECT IAcadSelectionSet 00f24024>
; Plot (RO) = #<VLA-OBJECT IAcadPlot 00f25f5c>
; PlotConfigurations (RO)=#<VLA-OBJECT IAcadPlotConfigurations 00f25ea4>
; Preferences (RO)=#<VLA-OBJECT IAcadDatabasePreferences 00f203b4>
; ReadOnly (RO) = 0
; RegisteredApplications(RO)=#<VLA-OBJECT AcadRegisteredApplications
00f259d4>
; Saved (RO) = 0
; SelectionSets (RO) = #<VLA-OBJECT IAcadSelectionSets 00f22764>
; TextStyles (RO) = #<VLA-OBJECT IAcadTextStyles 00f25794>
; UserCoordinateSystems (RO) = #<VLA-OBJECT IAcadUCSs 00f25554>
; Utility (RO) = #<VLA-OBJECT IAcadUtility 00f25384>
; Viewports (RO) = #<VLA-OBJECT IAcadViewports 00f252f4>
; Views (RO) = #<VLA-OBJECT IAcadViews 00f25014>
; Width = 978
; WindowState = 3
; WindowTitle (RO) = "Drawing1.dwg"
;Methods supported:
; Activate ()
; AuditInfo (1)
; Close (2)
; CopyObjects (3)
; EndUndoMark ()
; Export (3)
; GetVariable (1)
; HandleToObject (1)
; Import (3)
; LoadShapeFile (1)
; New (1)
; ObjectIdToObject (1)
; Open (1)
; PurgeAll ()
; Regen (1)
; Save ()
; SaveAs (2)
; SendCommand (1)
; SetVariable (2)
; StartUndoMark ()
; Wblock (2)
```

Wow! If you're not used to working with Visual LISP or ActiveX, you should be able to see how powerful this is for you as a software developer. A careful review of the above results will reveal all of the things you can get and modify with respect to a given document. This was simply not possible with AutoLISP prior to Visual LISP. As you can see, you now have direct access to all the tables, actually collections, within this document, as well as system variables, methods and so forth.

Note that the Plot property is not a method at all. This is because the Plot property of the Document object, is actually a pointer to the Plot object. The Plot object is how you configure and execute printing within the ActiveX world in AutoCAD.

Let's use some of these methods in conjunction with the Documents collection to see how we can iterate all opened documents and perform a simple task on each one. How about if we want to run the AUDIT command on all of our opened drawings:

```
(defun AllDocs-Audit ( / docs)
  (vlax-for dwg (vla-get-documents (vlax-get-acad-object))
    (vla-auditinfo dwg T); T denotes fix errors = Yes
  )
)
```

We can also Save all opened documents as follows:

```
(defun AllDocs-Save ( / docs)
  (vlax-for dwg (vla-get-documents (vlax-get-acad-object))
    (vla-save dwg)
  )
)
```

How about running the PURGE command on all opened documents?

```
(defun AllDocs-Purge ( / docs)
  (vlax-for dwg (vla-get-documents (vlax-get-acad-object))
    (vla-purgeall dwg)
  )
)
```

Warning!

Be careful when iterating the Documents collection to perform certain tasks. Keep in mind that LISP operates in a document context, not an application context. This means that while you can operate on other opened documents within the same application namespace, you are still calling your changes from the document in which your function was executed. You should always try to have your code return focus to the originating document when it completes or when it fails due to an error. This is especially important if you plan to implement reactors and invoke their callbacks from other drawing sessions than the one in which you instantiated the reactors and callbacks.

Chapter 19 – The Preferences Objects

That's not a misprint, the title of this Chapter mentions Objects (plural). The reason is simply that there are two main Preferences collection objects within AutoCAD. The first is the AcadPreferences collection object, which applies to AutoCAD itself. The second is the Document Preferences collection object, also called the DatabasePreferences collection object. The latter applies only to Document objects, not to AutoCAD.

To give you a better example, if you open the OPTIONS dialog form and browse throughout all the available tabs, you'll notice that many settings have a small drawing icon symbol beside them. This denotes settings that are saved to the drawing only and do not carry across to other drawing sessions. These items are actually part of the DatabasePreferences collection.

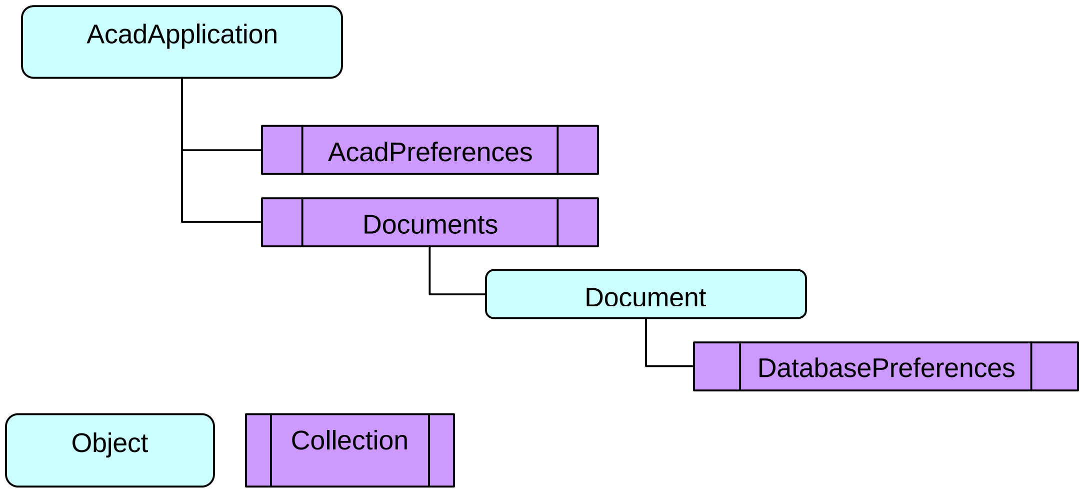


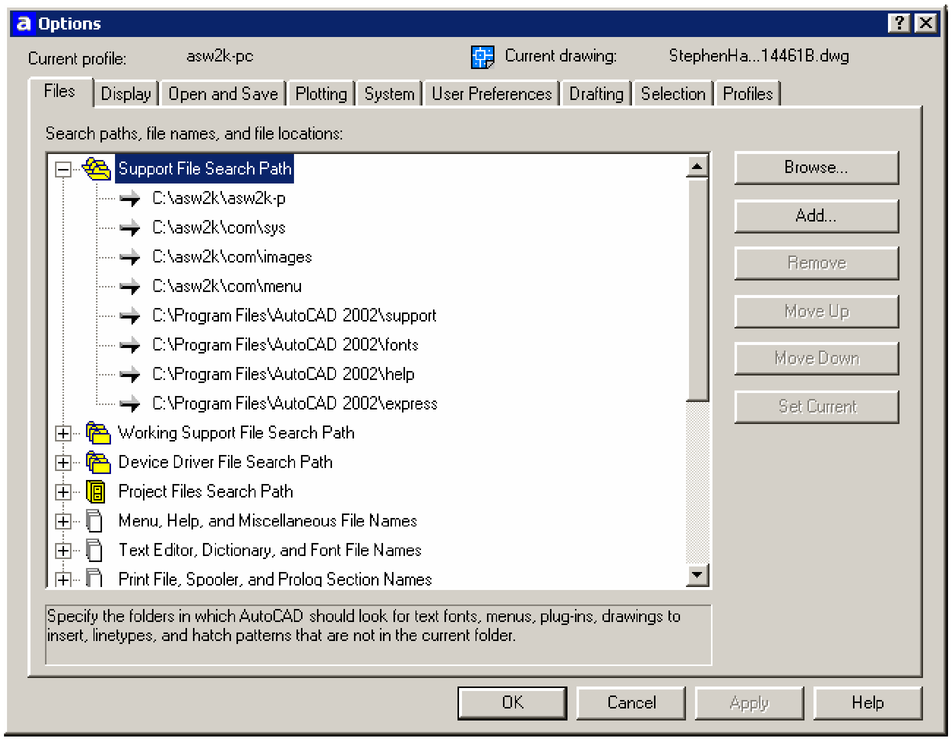
Figure 19-1 – The AcadPreferences and DatabasePreferences collection objects

The AcadPreferences Collection Object

The AcadPreferences collection is actually a container for other collections, each having their own objects. The graphical nature of the OPTIONS dialog form is not always correct in how it represents these collections and should not be used as a guide or map for understanding these collections. Some objects have different names and some are shown as collections in OPTIONS but stored as a single object in the AcadPreferences object.

For example, the Support Path list shown in the OPTIONS File tab is displayed as a list of sub items, string values of pathnames. This could easily be mistaken for being a collection of paths. But in actuality it is a single string value with a semi-colon delimiter between each path value and stored as the SupportPath property within the Files collection. Confusing? It can be. Another example is the Data Sources Location path setting on the Files tab of the OPTIONS dialog form. This is actually stored as the

WorkSpace property under the Files collection, not as DataSourcesPath or something more intuitively named.



Search paths, file names, and file locations:

- Support File Search Path
  - C:\asw2k\asw2k-p
  - C:\asw2k\com\sys
  - C:\asw2k\com\images
  - C:\asw2k\com\menu
  - C:\Program Files\AutoCAD 2002\support
  - C:\Program Files\AutoCAD 2002\fonts
  - C:\Program Files\AutoCAD 2002\help
  - C:\Program Files\AutoCAD 2002\express
- Working Support File Search Path
- Device Driver File Search Path
- Project Files Search Path
- Menu, Help, and Miscellaneous File Names
- Text Editor, Dictionary, and Font File Names
- Print File, Spooler, and Prolog Section Names

Specify the folders in which AutoCAD should look for text fonts, menus, plug-ins, drawings to insert, linetypes, and hatch patterns that are not in the current folder.

Figure 19-2 – The AutoCAD OPTIONS dialog form, Files tab displayed.

The AcadPreferences object has nine collection objects within it, one Property (the Application property) and no Methods. The AcadPreferences collections roughly correspond to the OPTIONS dialog tabs, sort of. Below is a list of the collection names and their corresponding tabs in the OPTIONS dialog form:

- Display = Display
- Drafting = Drafting
- Files = Files
- OpenSave = Open and Save
- Output = Plotting
- Profiles = Profiles
- Selection = Selection
- System = System
- User = User Preferences

Figure 19-3 shows a dump of the AcadPreferences object using (vlax-dump-object (vla-get-preferences (vlax-get-acad-object)) T).

```
(vlax-dump-object (vla-get-preferences (vlax-get-acad-object)) t)
; IAcadPreferences: This object specifies the current AutoCAD settings
; Property values:
;  Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;  Display (RO) = #<VLA-OBJECT IAcadPreferencesDisplay 04c5df7c>
;  Drafting (RO) = #<VLA-OBJECT IAcadPreferencesDrafting 04c5df78>
;  Files (RO) = #<VLA-OBJECT IAcadPreferencesFiles 04c5df80>
;  OpenSave (RO) = #<VLA-OBJECT IAcadPreferencesOpenSave 04c5df84>
;  Output (RO) = #<VLA-OBJECT IAcadPreferencesOutput 04c5df88>
;  Profiles (RO) = #<VLA-OBJECT IAcadPreferencesProfiles 04c5df8c>
;  Selection (RO) = #<VLA-OBJECT IAcadPreferencesSelection 04c5df90>
;  System (RO) = #<VLA-OBJECT IAcadPreferencesSystem 04c5df94>
;  User (RO) = #<VLA-OBJECT IAcadPreferencesUser 04c5df98>
; No methods
```

Figure 19-3 – Dump of the AcadPreferences collection object.

To go deeper, we'll examine the Files collection to see what it contains…

```
(vlax-dump-object (vla-get-files (vla-get-preferences (vlax-get-acad-object))) T)
; IAcadPreferencesFiles: This object contains the options from the Files tab on
the Options dialog
; Property values:
;  AltFontFile = "simplex.shx"
;  AltTabletMenuFile = ""
;  Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;  AutoSavePath = "C:\\TEMP\\"
;  ConfigFile (RO) = "C:\\Program Files\\AutoCAD 2002\\acad2002.cfg"
;  CustomDictionary = "C:\\Program Files\\AutoCAD 2002\\support\\sample.cus"
;  DefaultInternetURL = "http://amsecweb.amsec.com/cad/sw"
;  DriversPath = "C:\\Program Files\\AutoCAD 2002\\drv"
;  FontFileMap = "C:\\Program Files\\AutoCAD 2002\\support\\acad.fmp"
;  HelpFilePath = "C:\\Program Files\\AutoCAD 2002\\help\\acad.chm"
;  LicenseServer (RO) = ""
;  LogFilePath = "C:\\program files\\autocad 2002\\"
;  MainDictionary = "enu"
;  MenuFile = "C:\\Program Files\\AutoCAD 2002\\support\\acad"
;  ObjectARXPath = ""
;  PostScriptPrologFile = ""
;  PrinterConfigPath = "C:\\Program Files\\AutoCAD 2002\\plotters"
;  PrinterDescPath = "C:\\Program Files\\AutoCAD 2002\\drv"
;  PrinterStyleSheetPath = "C:\\Program Files\\AutoCAD 2002\\Plot Styles"
;  PrintFile = "."
;  PrintSpoolerPath = "C:\\TEMP\\"
;  PrintSpoolExecutable = ""
;  SupportPath= "C:\\Program Files\\AutoCAD 2002\\support;C:\\Program
Files\\AutoCAD 2002\\fonts;C:\\Program Files\\AutoCAD 2002\\help;C:\\Program
Files\\AutoCAD 2002\\express"
;  TempFilePath = "C:\\TEMP\\"
;  TemplateDwgPath = "C:\\Program Files\\AutoCAD 2002\\template"
;  TempXrefPath = "C:\\TEMP\\"
;  TextEditor = "Internal"
;  TextureMapPath = "C:\\Program Files\\AutoCAD 2002\\textures"
;  WorkspacePath = "C:\\Program Files\\AutoCAD 2002\\Data Links"
; Methods supported:
;  GetProjectFilePath (1)
;  SetProjectFilePath (2)
```

Figure 19-4 – Dump of the Files collection object within AcadPreferences

Note that the Files collection has many Properties and only two methods available. Also, you'll notice here that the SupportPath setting shows the search path list as a single string with semi-colon delimiters between each path value.

The most important understanding to come away from this is that you can use the (vla-get-xxx) and (vla-put-xxx) functions to get and modify any properties shown anywhere throughout the AcadPreferences collection objects as long as they are not read-only (RO). Think about this for a few minutes and it should dawn on you that this exposes an enormous amount of power and flexibility to you as the developer. You can programmatically manipulate the AutoCAD configuration with very little effort. Taking this farther, when you begin working with Profiles, you will find this opens up a whole world of possibilities to manage desktops remotely in a networked environment.

Let’s demonstrate how this might be put to practical use with Visual LISP. Suppose you want to modify all the AutoCAD installations on your network to change a default path setting for where each client looks for Drawing Template files. Maybe you want them to all use a standard set of customized templates stored in a folder on a shared server over the network. To do this, you only need to push out a change to the TemplateDwgPath property under the Files collection. Sure, you could do this using a profile (.ARG file) or a registry hack, but deploying those are difficult without some additional tools or some scripting to help it work.

One solution is to push the update through AutoCAD using ActiveX with Visual LISP coding and the AcadPreferences object interface. Figure 19-5 shows an example function for doing just that and shows an example of how it might be used within a program function. This could be deployed by using a hook from within the acaddoc.lsp (s::startup) function that you could deploy to all clients once, and be able to deploy code changes from then on with very little effort. The code in Figure 19-5 could be loaded from the (s::startup) routine and fire off automatically after it loads on the client.

```
(defun UpdateTemplatePath (pathname)
  (vla-put-TemplateDwgPath
    (vla-get-Files
      (vla-get-AcadPreferences
        (vlax-get-acad-object)
      )
    )
    pathname
  )
)
(UpdateTemplatePath “J:\\acad\\configs\\templates”)
```
Figure 19-5 – Updating the TemplateDwgPath property via the AcadPreferences/Files interface

You might want to embellish this code a little to make it more robust and flexible. For example, you could add a check to make sure the existing path setting is not already correct before changing it, saving unnecessary work at the start of every drawing session on every client.

The DatabasePreferences Object

The DatabasePreferences object is a collection of preferences that apply to the active document only. As stated previously, they appear in the OPTIONS dialog with a small drawing icon symbol beside them to indicate this.

Below is a dump of the collection from a drawing session to show what items it contains:

```
Command: (setq dbprefs (vla-get-preferences activedocument))
Command: (vlax-dump-object dbprefs t)
; IAcadDatabasePreferences: This object specifies the current AutoCAD drawing
specific settings
; Property values:
;   AllowLongSymbolNames = -1
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;   ContourLinesPerSurface = 4
;   DisplaySilhouette = 0
;   Lineweight = -1
;   LineWeightDisplay = -1
;   MaxActiveViewports = 64
;   ObjectSortByPlotting = 0
;   ObjectSortByPSOutput = 0
;   ObjectSortByRedraws = 0
;   ObjectSortByRegens = 0
;   ObjectSortBySelection = -1
;   ObjectSortBySnap = -1
;   OLELaunch = 0
;   RenderSmoothness = 0.5
;   SegmentPerPolyline = 8
;   SolidFill = -1
;   TextFrameDisplay = 0
;   XRefEdit = -1
;   XRefLayerVisibility = -1
; No methods
```

Figure 19-6 – The DatabasePreferences collection object

You'll notice that there are no methods to this object. Just as with the AcadPreferences object, you can access and manipulate these properties using the same approach with VLISP. For example, to toggle Lineweight display on or off:

```
(vla-put-LineWeightDisplay activedoc :vlax-true);; turns LWT on
(vla-put-LineWeightDisplay activedoc :vlax-false);; turns LWT off
```

Reloading a Profile

AutoCAD R14.01 added a nice feature that allows you to specify a profile .ARG file in the shortcut parameters using the /p start-up switch. However, AutoCAD will not allow you to reload an .ARG file when the target profile name is already defined. AutoCAD sees the name in the ARG file and sees an existing profile already defined and simply points to the existing profile and ignores the ARG file. The only workarounds are to...

- Set the active profile to some other profile, and then delete the target profile. Then import the ARG file to redefine the profile again.

- Rename the existing profile and import the ARG file to define the profile again.

- Import the ARG file such that it defines a new (unique) profile name.

Why would you want to be able to "force reload" a profile? Well, there is also another undocumented "feature" of AutoCAD that affects network profile configurations in a bad way. What happens is that if your profile includes network pathing settings, and the network paths happen to be inaccessible at the point when a user launches AutoCAD with that profile, AutoCAD handily strips out the pathing entirely and continues on. Obviously, this is not good, especially since it will not add the pathing back in even when the network is back on line and accessible.

So, now what? Well, since you have the means to access and manipulate the AcadPreferences object through Visual LISP, you can easily work your way around this under the hood. Check out the example code functions below.

```
;;; Reloads a profile from an ARG file
;;; Replaces existing profile if defined
;;; Returns profile name if successful, otherwise returns nil

(defun Profile-Reload (name ARGname / bogus)
  (cond
    ( (and
        (Profile-Exists-p name)
        (findfile ARGname)
      )
      (if (/= (strcase name) (strcase (vla-get-ActiveProfile (AcadProfiles))))
        (Profile-Delete name)
        (progn
          (setq bogus "bogus")
          (Profile-Rename name bogus)
        )
      )
      (Profile-Import name ARGname)
      (vla-put-ActiveProfile (AcadProfiles) name)
      (if bogus (Profile-Delete bogus))
      name
    )
    ( (and
        (not (Profile-Exists-p name))
        (findfile ARGname)
      )
      (Profile-Import name ARGname)
      (vla-put-ActiveProfile (AcadProfiles) name)
      name
    )
    ( (not (findfile ARGname))
      (princ (strcat "\nCannot locate ARG source: " ARGname)) nil
    )
  )
)

;;; Renames an existing profile
;;; Returns new profile name if successful, otherwise returns nil

(defun Profile-Rename (from to / result)
  (if (Profile-Exists-p from)
    (if (not (Profile-Exists-p to))
      (cond
        ( (not
            (vl-catch-all-error-p
              (setq result
                (vl-catch-all-apply
                  'vla-RenameProfile
                  (list (AcadProfiles) from to)
                )
              )
            )
          )
          to ; Return new name if successful!
        )
      )
    ); end if
  ); end if
)

;;; Deletes an existing profile
;;; Returns T if successful, otherwise returns nil
```

```
(defun Profile-Delete (strName / result)
  (if (Profile-Exists-p strName)
    (cond
      ( (not
          (vl-catch-all-error-p
            (setq result
              (vl-catch-all-apply
                'vla-DeleteProfile
                (list (AcadProfiles) strName)
              )
            )
          )
        )
        T ; return T for success!
      )
    )
  ); endif
)

;;; Imports a profile from a given ARG file
;;; Returns profile name if successful, otherwise returns nil

(defun Profile-Import (argFile strName / result)
  (cond
    ( (findfile argFile)
      (cond
        ( (not
            (vl-catch-all-error-p
              (setq result
                (vl-catch-all-apply
                  'vla-ImportProfile
                  (list (AcadProfiles) strName argFile vlax-True)
                )
              )
            )
          )
          strName ; return new profile name if successful!
        )
      )
    )
  )
)

;;; Determine if profile name is already defined (exists)
;;; Returns T or nil

(defun Profile-Exists-p (name)
  (get-item (AcadProfiles) name)
)

;;; Return Profiles collection object

(defun AcadProfiles ()
  (vla-get-profiles (vla-get-preferences (vlax-get-acad-object)))
)
```

Last but not least, here is a simple function to return a list of all defined profile names:

```
(defun Profiles-ListAll ( / hold)
  (vla-GetAllProfileNames (AcadProfiles) 'hold)
  (if hold (vlax-SafeArray->List hold))
)
```

Chapter 20 – Menus and Toolbars

Just as AutoCAD drawing entities are part of the AutoCAD object model, so are menus and toolbars. Menus are organized into a root MenuGroup collection, containing one or more MenuGroup objects. Each MenuGroup object in turn contains a collection of "popmenus" (pull-down), and a collection of toolbars. You can add MenuGroups using the `(vla-load)` method on the MenuGroups object. This performs the same task as using the AutoCAD MENULOAD command.

The coolest thing about this is that you can add, modify and delete menu items and configurations programmatically. This enables you to build menus with complete control from within your VLISP programs. The only thing not exposed from the menugroups or menubar objects is the screen menu from ancient days. For this you must resort to the standard AutoLISP (menucmd) function to manipulate the screenmenu and the AcadPreferences collection to control the display (toggle it on or off).

The MenuBar Object

The MenuBar object contains all of the currently displayed pulldown or "pop" menu items in the AutoCAD session. It is a member of the AcadApplication object. If you have stub menus loaded that have any pop menu groups spliced in with the AutoCAD pop menus, the MenuBar object will return all of them in the order from left to right in the collection.

```
(setq acad-app (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00a8a730>

(setq mbar (vla-get-menubar acad-app))
```

To access individual pop menus, use the Item method as follows:

```
(setq popmenu1 (get-item mbar 0))
#<VLA-OBJECT IAcadMenuBar 00e8ae24>
```

To see more menubar info, dump the object as follows:

```
Command: (vlax-dump-object mbar T)
; IAcadMenuBar: A collection of PopupMenu objects representing the
current
AutoCAD menu bar
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;   Count (RO) = 14
;   Parent (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
; Methods supported:
;   Item (1)
```

Getting MenuBar Items

To access a MenuBar item or check if a popmenu exists in the collection you can iterate the collection. To check for a particular popmenu, search for a matching name value in the MenuBar collection as follows:

```
(defun PopMenu-MenuBar-p (name / mbar i found)
  (setq mbar (vla-get-menubar (vlax-get-acad-object)) i 0)
  (while (and (not found) (< i (1- (vla-get-count mbar))))
    (if (= (strcase name) (strcase (vla-get-name (get-item mbar i))))
      (setq found T)
    )
    (setq i (1+ i))
  )
  (vlax-release-object mbar)
  found
)
```

If you know the name of the pop menu, you can access it directly using Item and the name property in string form as follows:

```
(setq popmenu (get-item mbar “&File”))
```

Tip!

Be aware that the name property includes the mnemonic character & as part of the name string. If you try to fetch the menu simply by the logical name of “File”, the `(vla-item)` method will fail to return it from the MenuBar object. This is true in general for using the Item method with any collection where you are fetching by string name values.

Inserting PopMenus into the MenuBar collection

To insert a popmenu of a loaded menugroup into the MenuBar collection, use the `(vla-InsertInMenuBar)` method of the popmenu object itself.

```
(defun PopMenu-Insert (mgroup name loc / pmnu)
  (if (not (PopMenu-MenuBar-p name))
    (if (setq pmnu (PopupMenu mgroup name))
      (progn
        (vla-insertinmenubar pmnu loc)
        (vlax-release-object pmnu)
        T
      )
      (princ (strcat "\nMenugroup or popmenu not loaded: " name))
    )
    (princ (strcat "\nPopmenu already loaded: " name))
  )
)
```

Removing PopMenus from the MenuBar collection

To remove a named popmenu from the MenuBar collection, use the `(vla-removefrommenubar)` method of the popmenu object itself.

The MenuGroups Collection Object

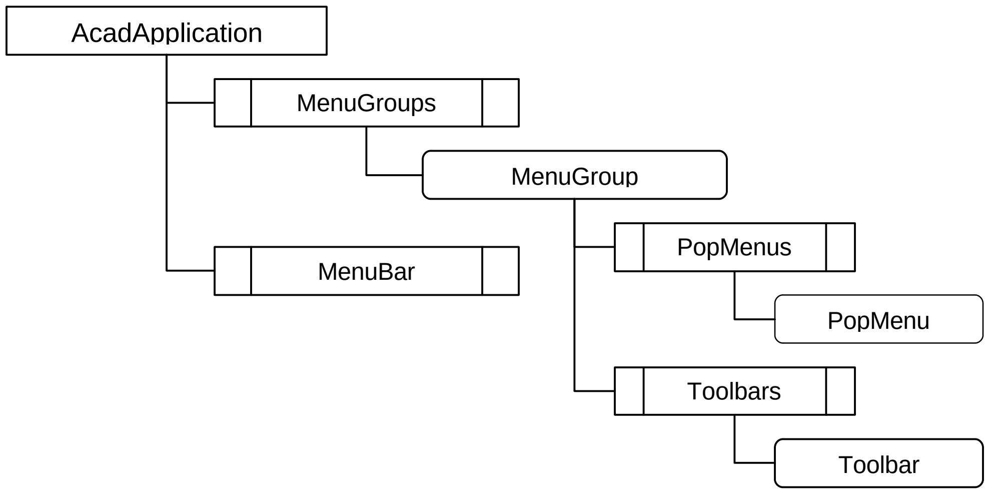


Figure 20-1 – The MenuGroup collection object

The MenuGroups collection object contains all the menugroups found in the AutoCAD session. Each menugroup is a source menu that has been loaded. Normally, you will see the menugroup “Acad”, but you might also see “Express” for the Express Tools menugroup if it has been installed.

```
(defun MenuGroups-ListAll ( / out)
  (vlax-for each (vla-get-menugroups acad-app)
    (setq out (cons (vla-get-name each) out))
  )
  out
)
```

To add a new menugroup to the menugroups collection, you use the Add or Load methods and specify the appropriate arguments. To remove a menugroup, you must first get the menugroup object and then invoke its Delete method.

The MenuGroup Object

The menugroup object contains all the popmenus and toolbars for a given menugroup. It is a member of the MenuGroups collection object. To get the Acad menugroup, use the following:

```
(setq mgroups (vla-get-menugroups acad-app))
#<VLA-OBJECT IAcadMenuGroups 01433208>

(setq mg-acad (get-item mgroups “Acad”))
#<VLA-OBJECT IAcadMenuGroup 00e9b38c>
```

The PopMenus Object

The PopMenus collection object contains all the popmenus, or pull-down menus, for a given menugroup. To access the popmenus collection for the Acad menugroup, use the following:

```
(setq pmnu (vla-get-popmenus mg-acad))
```

A dump of the popmenus object shows its properties and methods:

The PopMenu Object

The PopMenu object represents a single popmenu or pull-down menu. You can use the PopMenu object to insert itself into the MenuBar object as well as to access its internal objects, properties and methods.

The Toolbars Collection Object

The Toolbars object is a collection of all toolbars for a given menugroup. Use the (vla-get-toolbars) method to access the root of this collection from the menugroup object. For example, to get the Toolbars collection for the Acad menugroup, use the following:

```
(setq tbars (vla-get-toolbars mg-acad))
#<VLA-OBJECT IAcadToolbars 00efa7c4>
```

You may want return a list of all the toolbar names for a given toolbars collection. To do this you simply need to iterate through the toolbars and return a list of their respective Name property values…

```
(progn
  (vlax-for each tbars
    (setq out (cons (vla-get-name each) out))
  )
  (if out (reverse out))
)
```

The Toolbar Object

The Toolbar object represents a single toolbar and it buttons. The buttons are contained as a collection which can be iterated using the `(vla-item)` method, instead I'll use the `(get-item)` function suggested earlier. For example, to access the "Dimension" toolbar object from the Acad menugroup, use the following:

```
(setq tb1 (get-item tbars "Dimension"))
#<VLA-OBJECT IAcadToolbar 00eb6a7c>

Command: (vlax-dump-object tb1 T)
; IAcadToolbar: An AutoCAD toolbar
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00a8a730>
;   Count (RO) = 22
;   DockStatus (RO) = 4
;   FloatingRows = 1
;   Height (RO) = 52
;   HelpString = "Dimension Toolbar"
;   LargeButtons (RO) = 0
;   left = 255
;   Name = "Dimension"
;   Parent (RO) = #<VLA-OBJECT IAcadToolbars 00efa7c4>
;   TagString (RO) = "ID_TbDimensi"
;   top = 218
;   Visible = -1
;   Width (RO) = 565
; Methods supported:
;   AddSeparator (1)
;   AddToolbarButton (5)
;   Delete ()
;   Dock (1)
;   Float (3)
;   Item (1)
```

The following example functions demonstrate how to get the Toolbars collection from a given menugroup object, and how to get a specified toolbar object by name from a specified menugroup object.

```
(defun get-MenuGroups ()
  (vla-get-menugroups (vlax-get-acad-object))
)

(defun get-MenuGroup (name)
  (if (menugroup name)
    (get-item (get-MenuGroups) name)
  )
)

(defun get-Toolbars (mgroup / mg result)
  (if (setq mg (get-MenuGroup mgroup))
    (progn
      (setq result (vla-get-Toolbars mg))
      (vlax-release-object mg)
     )
   )
  result
)

(defun get-Toolbar (mgroup name / tbs result)
  (if (setq tbs (get-Toolbars mgroup))
    (progn
      (setq result (get-item tbs name))
      (vlax-release-object tbs)
    )
  )
  result
)
```

The following example function docks a specified toolbar to the Left, Right, Top or Bottom. If the *side* argument is not “LEFT”, “RIGHT”, “TOP” or “BOTTOM” then it defaults to “LEFT”. The argument is not case sensitive.

```
(defun Toolbar-Dock (mgroup name side / tb loc)
  (cond
    ( (= (strcase side) “LEFT”)  (setq loc acToolbarDockLeft))
    ( (= (strcase side) “RIGHT”) (setq loc acToolbarDockRight))
    ( (= (strcase side) “TOP”)   (setq loc acToolbarDockTop))
    ( (= (strcase side) “BOTTOM”)(setq loc acToolbarDockBottom))
    ( T (setq loc acToolbarDockLeft))
  )
  (if (setq tb (get-Toolbar mgroup name))
    (progn
      (vla-Dock tb loc)
      (vlax-release-object tb)
    )
    (princ (strcat "\nToolbar (" name ") not found."))
  )
)
```

The following example function floats a toolbar at a specified location (x and y offset values from top-left of screen). The arguments are the menugroup name, the toolbar name, y-coordinate, x-coordinate and toolbar row layout. The x and y coordinates are from the top-left corner of the screen. This is a Windows standard practice for dialog forms and toolbars. This function ignores toolbars that are hidden.

```
(defun Toolbar-Float (mgroup name top left rows)
  (if (setq tb (get-Toolbar mgroup name))
    (if (= (vla-get-Visible tb) :vlax-True)
      (progn
        (vla-Float tb top left rows)
        (vlax-release-object tb)
        1 ;; float and visible
      )
      -1 ;; toolbar not visible
    )
    0 ;; toolbar not found
  )
)
```

Creating a Toolbar

Let’s assemble some code covered above and add some new ingredients to make a new toolbar and assign a few buttons to it. In this case, we’ll add a new toolbar to the ACAD menugroup and name it MYTOOLBAR. The first function adds a button object to a toolbar object with some supplied property values. In this example, I use the same bitmap property for both large and small icon bitmap properties.

```
(defun Toolbar-AddButton
  (tb name macro bitmap1 tagstring helpstring / newButton index)
  (setq index (vla-get-Count tb))
  (cond
    ( (setq newButton
        (vla-AddToolbarButton tb
          (vlax-make-variant index vlax-vbInteger)
          name helpstring macro
        )
      )
      (vla-put-TagString newButton tagstring)
      (vla-SetBitMaps newButton bitmap1 bitmap1)
      newButton
    )
  )
)
```

Now, we'll see how to create a new toolbar and assign a new button to it. This function will create a new toolbar named "MyToolbar" and add one button to it that invokes the LINE command. Load the sample file Toolbars.lsp from the book CD samples and run the function (toolbar-make) at the AutoCAD command prompt.

```
(defun Toolbar-Make ( / tb)
  (cond
    ( (setq tb (vla-add (get-toolbars "acad") "MyToolbar"))
      (if (Toolbar-AddButton tb
        "Line" "\003\003\020\nLine"
        "ICON_16_LINE" "MyButton001" "Draws a line: LINE"
        )
        (alert "I just added a button to my toolbar!")
        (alert "Uh oh! ...")
      )
      (vlax-release-object tb)
    )
  )
)
```

You can continue much further with this by tapping the other methods such as Add, Delete and so on to build out your toolbars. You can also manipulate toolbar row configurations, change button ordering and hide or display the toolbar. When you combine this with your program code you can create some very sophisticated menu management features.

Tip!

One very common request I get from instructors involves how to automatically "reset" menu and toolbar configurations after students mess them up during a class. Profiles are one way, but even profiles can be unreliable when it comes to menus, because menus rely upon MNS and MNC files as well as the registry for their configurations. The more reliable method is to combine the examples show above into a comprehensive menu manager that steps through all the menu items and sets them to a desired property setting (location, display, docking, etc.).

Chapter 21 – Interfacing with Other Applications

While Visual LISP is a bit more cumbersome to use for dealing with external applications than VBA, it does have the capability to do some very powerful things. While it is very common to use ObjectARX applications from within Visual LISP, such as DOSlib, it is only a small taste of what more can be done. Some examples might be passing data between AutoCAD and Microsoft Office applications, sending E-mail messages in the background using CDONTS, and performing specialized desktop, file, folder and network tasks using the Windows Scripting Host.

Microsoft Excel

The example code below shows how to initialize the Excel Type Library (for Excel 2000 or XP) and create a new Excel spreadsheet file in Excel from Visual LISP. Be careful of one particular aspect of Visual LISP: TypeLib interfaces. Why would I say this? Well, while VB and VBA give you the nice auto-complete feature called Intellisense®, Visual LISP does not.

When working in VB or VBA and setting a *reference* to a component library, it takes care of mapping the syntax awareness and pop-up help strings. Visual LISP does not. Even though you may be familiar with Excel 2000 from a Visual LISP standpoint, don't assume everything is the same when it comes to Excel 10 (also called XP).

There's not enough space to cover this in detail, but be assured that it's well worth the time to investigate changes anytime you intend to use your code with a newer version of an external application.

```
(defun Excel-TypeLib-2000 ( / sysdrv officepath)
  (setq sysdrv (getenv "systemdrive"))
  (setq officepath (strcat sysdrv "\\program files\\microsoft office\\office"))
  (findfile (strcat officepath "\\excel9.olb"))
)
(defun Excel-TypeLib-XP ( / sysdrv officepath)
  (setq sysdrv (getenv "systemdrive"))
  (setq officepath (strcat sysdrv "\\program files\\microsoft offices\\office10"))
  (findfile (strcat officepath "\\excel.exe"))
)
(defun Excel-Load-TypeLib ( / tlbfile tlbver out)
  (cond
    ( (null msxl-xl24HourClock)
      (if (setq tlbfile (Excel-TypeLib-2000))
        (progn
          (vlax-import-type-library
            :tlb-filename      tlbfile
            :methods-prefix    "msxl-"
            :properties-prefix "msxl-"
            :constants-prefix  "msxl-"
          )
          (if msxl-xl24HourClock (setq out T))
        )
      )
    )
    ( T (setq out T) )
  )
  out
)
(defun Excel-New-Spreadsheet (dmode / appsession result)
  (princ "\nCreating new Excel Spreadsheet file...")
  (cond
    ( (vl-catch-all-error-p
        (setq appsession
          (vl-catch-all-apply
            'vlax-create-object
           '("Excel.Application")
          )
        )
      )
      (vl-exit-with-error
        (strcat "Error: " (vl-catch-all-error-message appsession))
      )
    )
    ( T
      (princ "\nOpening Excel Spreadsheet file...")
      (cond
        ( (vl-catch-all-error-p
            (setq result
              (vl-catch-all-apply
                'vlax-invoke-method
                (list
                  (vlax-get-property appsession "Workbooks")
                  "Add"
                )
              )
            )
          )
          (princ (strcat "\nError: " (vl-catch-all-error-message result)))
        )
        ( T
          (if (= (strcase dmode) "SHOW")
            (vla-put-Visible appsession 1)
            (vla-put-Visible appsession 0)
          )
        )
      )
    )
  )
  appsession
)
```

You should pay special attention to the section in the () function above that defines the type library interfaces. This is not very clearly documented actually, but the string prefixes assigned to the properties, methods and constants is arbitrary. In this example, I used the same value for all three, but other examples you'll find will use unique prefixes such as `msxp-` `msxm-` and `msxc-` to differentiate each of the types of interface objects.

```
(vlax-import-type-library
  :tlb-filename      tlbfile
  :methods-prefix    "msxl-"
  :properties-prefix "msxl-"
  :constants-prefix  "msxl-"
)
```

Theoretically, you could also forego assigning a prefix by using "" for each property in the declaration expression above. But doing so will make it difficult to work with multiple application type library interfaces, such as Word and Excel, that might be used within the same Visual LISP code. Yes, you can define and use as many type library interfaces as you need to do the job. It's usually better to modularize your Visual LISP code to avoid this if possible, and keep each type library reference isolated. Debugging and testing will be much easier to manage by keeping things orderly and organized.

Warning! There is a known defect in AutoCAD 2000-2002 when releasing objects created using `(vlax-get-or-create-object)` as well as `(vlax-create-object)` whereby the external process is not terminated when the object is released.

You can verify this by using the Windows Task Manager and watching the Processes list for a given application. For example, if you open a session of Excel 10 (part of Office XP) and call the Quit method of the Excel.Application object, you would expect that after you release the object in VLISP that the process would terminate, but it usually will not. The following example code can be used to test this on your computer:

```
(defun excel-test ( / xlapp)
  (cond
    ( (setq xlapp (vlax-create-object “Excel.Application”))
      (vlax-put-property xlapp “Visible” T); show Excel
      (vlax-invoke-method xlapp “Quit”); close Excel
      (vlax-release-object xlapp); release object
      (gc); force garbage collection
    )
    ( T (princ “\nUnable to open Microsoft Excel.”) )
  )
)
```

Load the above code into the VLIDE edit window and load it into AutoCAD. Open the Windows Task Manager and pick the Processes list tab. Go back to AutoCAD and run the function (excel-test) and watch the Task Manager process list for Excel.exe to appear in the list.

Proper behavour would be that the Excel.exe process would appear and then disappear, which it may do in your case. But in the majority of cases it will not disappear. The problem this creates is if you attempt to reopen a given spreadsheet and the Excel process has not let go of it. The spreadsheet file may often be opened in Read-Only mode since it thinks someone else has the file already opened.

Autodesk suggests using (gc) after releasing such objects to force a termination, however, (gc) simply places a call to the garbage collection service on a stack which is managed by the Windows resource services. In other words, regardless of how you try to terminate the session from VLISP, it may often not terminate at all. Be careful when manually terminating the process using Task Manager as it can often break the RPC channel to AutoCAD and make subsequent calls to Excel fail with errors.

Windows Scripting Host

Microsoft’s Windows Scripting Host (WSH) is a powerful, yet often overlooked free service available as part of Windows 98/ME/2000/XP operating systems. It is also available for free download to install on Windows 95 and NT 4.0 systems. Basically, WSH is a script engine that runs scripts either from a command line interface or from a GUI interface. The command line interface command is CSCRIPT, while the GUI interface command is WSCRIPT. You can view the available runtime options by typing `CSCRIPT /?`, or `WSCRIPT /?` at the Windows Command Shell (DOS Window).

The example below demonstrates how to use the WSH Shell object to create shortcuts in the Favorites collection. You can also access the Desktop and Start Menu shortcuts repositories, for both the current user and the AllUsers group profiles (depending upon the local rights of the current user).

```
(defun AddFavoritesShortcut
  (target title / oWsh spfolders favorites shortcut)
  (cond
    ( (setq oWsh (vlax-create-object "WScript.Shell"))
      (setq spfolders (vlax-get-property oWsh "SpecialFolders")
            favorites (vla-item spfolders "Favorites")
             shortcut (vlax-invoke-method oWsh
                        “CreateShortcut”
                        (strcat favorites “\\” title “.lnk”)
                      )
      )
      (vlax-put-property shortcut "TargetPath" target)
      (vlax-invoke-method shortcut "Save")
      (vlax-release-object oWsh)
      (gc);; forced garbage collection after object release
      (princ "\nShortcut created in Favorites...")
    )
    ( T (alert "Failed to obtain shortcut class object..."))
  )
)
(defun C:FAV ( / target name)
  (setq target (getstring “\nURL for Favorite shortcut: “)
        name   (getstring t “\nName for Favorite: “)
  )
  (AddFavoriteShortcut target name)
  (princ)
)
```
Figure 22-1 – Using WSH to add a Favorites shortcut link

The FileSystem Object

The FileSystem object is a powerful tool for interfacing with, and manipulating files, and folders through the Windows operating system. For example, we can use it to iterate all drive mappings and return a list of drive mappings.

```
(defun ListDriveMappings (/ fso drive drives lst pth grp)
  (setq fso (vlax-Create-Object "Scripting.FileSystemObject"))
  (vlax-for drive (setq drives (vlax-get-Property fso "Drives"))
    (setq ltr (strcat (vlax-get-property drive "DriveLetter") ":")
          pth (vlax-get-property drive "ShareName")
          grp (cons ltr pth)
          lst (cons grp lst)
    )
  )
  (vlax-Release-Object drives)
  (vlax-Release-Object fso)
  (reverse lst)
)
```
Figure 22-2 – Using the FileSystem object to list drive mapping properties

The above function returns a list of drive mappings in paired sub-lists such as ((“C:” . “”) (“D:” . “”) (“F:” . “\\\\server\\share”) …). This can be useful for building lists and for validating a users’ configuration to support your application drive mapping needs.

We can also use the FileSystem object to see if a particular UNC path is mapped as a drive letter, and if so, return the actual drive letter.

```
(defun get-MappedShare (share / fso drives drive letter)
  (setq fso (vlax-create-object "Scripting.FileSystemObject"))
  (vlax-for drive (setq drives (vlax-get-property fso “Drives”))
    (if
      (=
        (strcase (vlax-get-property drive "ShareName"))
        (strcase share)
      )
      (setq letter (strcat (vlax-get-property drive "DriveLetter") ":"))
    )
  )
  (vlax-release-object drives)
  (vlax-release-object fso)
  letter
)
```

Figure 22-3 – Using the FileSystem object to resolve a UNC path to a drive letter mapping

Using the above example, you can resolve UNC paths to actual drive letters if they have been mapped by the current user. The syntax is as follows:

```
(get-MappedShare “\\\\myserver\\myshare”) could return “F:”
```

The FileSystem object provides many other methods and accesses to various object properties. For instance, you can copy, rename and delete files and folders. You can even use it to copy a file directly to a named port, such as when performing direct-port printing with plot files:

```
(defun CopyFileToLPT1 (filename / file fso)
  (setq fso (vlax-create-object "Scripting.FileSystemObject"))
  (setq file (vlax-invoke-method fso "GetFile" filename))
  (vlax-invoke-method file "Copy" "LPT1")
  (vlax-release-object file)
  (vlax-release-object fso)
)
```

Figure 22-4 – Using the FileSystem object to copy a file to the LPT1 port

Windows Messaging and CDONTS

CDONTS is an API provided by Microsoft Windows that enables applications to perform various messaging functions programmatically. A subset of CDO, *collaboration data objects*, CDONTS is a stripped-down, but very useful collection of functions, properties and methods which can be derived from and used to create, and send messages with various options.

Using an example from an Active Server Page (ASP) script, you can see how wecan create an instance of a CDONTS object and use it to send a message directly from the web server. This requires that the web server also has SMTP mail services running.

```
<%
Set objMail=CreateObject("CDONTS.NewMail")
objmail.From="davidmstein@yahoo.com"
objmail.To="davidmstein@yahoo.com"
objmail.Subject="CDONTS Email Test Message"
objmail.Body=”I love your book!”
objmail.Send
Set objMail=Nothing
%>
```

Figure 21-3 – ASP CDONTS example code.

The code in Figure 21-3 performs a simple email send from within an ASP web page when loaded and run from a Microsoft IIS web server. Ignoring the syntax and specifics of ASP code, you should be able to see that the first thing you do is create an object instance of the CDONTS.NewMail class to use for making and sending your email message.

To do this in a Visual LISP environment, you would use something like the translated code in Figure 21-4.

```
;; No error checking provided!

(defun SendMail ( / cdoMail)
  (cond
    ( (setq cdoMail (vlax-create-object "CDONTS.NewMail"))
      (vlax-put-property cdoMail 'From “davidmstein@yahoo.com”)
      (vlax-put-property cdoMail 'To “davidmstein@yahoo.com”)
      (vlax-put-property cdoMail 'Subject “CDONTS Test Message”)
      (vlax-put-property cdoMail 'Body “I love your book!”)
      ; Important property: 0=low, 1=normal, 2=high
      (vlax-put-property cdoMail 'Importance 1)
      (vlax-invoke-method cdoMail 'Send)
      (vlax-release-object cdoMail)
      (setq cdoMail nil)
    )
  )
  (princ)
)
```

Figure 21-4 – Visual LISP CDONTS example code

The code example in Figure 21-4 does not use any TypeLib interfaces, which is why the Importance property is assigned an integer value instead of something more intuitive like *cdo-Normal*.

The first line requests an object to be created from the CDONTS.NewMail class. Once that succeeds, the object is assigned properties for the From, To, Subject, Body and Importance values. Then the Send method is called to release the message. Finally, the object is released and set to *nil* to clear it from memory.

If you are familiar with the CDO class object, you might want to consider using it instead of CDONTS, as it provides much more granularity and flexibility.

Windows Management Instrumentation (WMI)

The Microsoft Windows Management Instrumentation (WMI) service is an abstraction layer that provides programmatic interfacing with system resource data. This includes hardware and software but also includes security and security context features. For a quick example of what WMI enables, open the Computer Management utility in Windows 2000 or Windows XP. Just about everything you can find in that collection of information is exposed by the WMI interfaces. WMI is accessible by any ActiveX programming language, including Visual LISP.

Invoking WMI from within Visual LISP is painful stuff. This doesn't mean that you can't make use of its mighty power from VLISP though. One very painless route is to invoke WMI operations from something that is built to handle them: VBScript. This makes it possible to place the code into a VBS file and execute it using WSH via the CSCRIPT command.

Let's start by building a WMI script in VBScript named ClearLogs.VBS. This script will clear all the local event logs and display a success or failure message at the end. Be careful if you enter this code manually in an editor! Not only are the property names rather long, but the wrapping caused by this printing produces incorrect results. Refer to the sample files included with this book to get the source code for this and other code shown throughout this book. The symbol ↘ indicates a word-wrap that should not be used when coding, but was required due to book formatting limitations.

```
Dim LogFileSet1, LogFileSet2, LogFileSet3, LogFile, RetVal, strInfo
strInfo = ""

Set LogFileSet1 = ↘
GetObject("winmgmts:{impersonationLevel=impersonate,(Backup)}").ExecQuery("SELECT * FROM
Win32_NTEventLogFile WHERE LogfileName='Application'")

For each Logfile in LogFileSet1
  RetVal = LogFile.ClearEventlog()
  If RetVal = 0 Then
    strInfo = strInfo & "The Application Log Has Been Cleared" & vbCrLf
  End If
Next

set LogFileSet2 = ↘
GetObject("winmgmts:{impersonationLevel=impersonate,(Backup)}").ExecQuery("SELECT * FROM
Win32_NTEventLogFile WHERE LogfileName='System'")

For each Logfile in LogFileSet2
  RetVal = LogFile.ClearEventlog()
  If RetVal = 0 Then
    strInfo = strInfo & "The System Log Has Been Cleared" & vbCrLf
  End If
Next

set LogFileSet3 = ↘
GetObject("winmgmts:{impersonationLevel=impersonate,(Backup)}").ExecQuery("SELECT * FROM
Win32_NTEventLogFile WHERE LogfileName='System'")

For each Logfile in LogFileSet3
```

```
  RetVal = LogFile.ClearEventlog()
  If RetVal = 0 Then
    strInfo = strInfo & "The Security Log Has Been Cleared"
  End If
Next
MsgBox strInfo, , "Event Log Status"
```

Now, to call this script from Visual LISP open a shell process and execute a concatenated request using CSCRIPT.

```
(setq scriptfile “c:\\myscripts\\clearlogs.vbs”)
(startapp (strcat “CSCRIPT.EXE //nologo “ scriptfile))
```

This example only barely covers WSH and WMI capabilities. You can combine such powerful features as Remote Scripting, Network Scripting and WMI to perform some incredible feats of daring that no manager could resist rewarding you with lavish pay increases and stock options. OK, so that’s going a bit far, but trust me, when you start combining Visual LISP with other language tools and interfaces, there’s no end in sight.

For more information on scripting using Microsoft VBScript or the Windows Scripting Host, check out http://www.microsoft.com/scripting , as well as great web sites like http://www.swynk.com , http://www.adminscripts.net and http://www.15seconds.com .

Working with Services

Tip!

When invoking Windows services, you normally use the GetObject method (vlax-get-object) with the explicit prog-id identifier. One problem exists in that Visual LISP cannot invoke certain services using this interface. Examples include the LanmanServer service, and Windows Management Instrumentation (WMI) mentioned above.

The only available workaround at the time of this publication is to provide a “wrapper”, or intermediate component to perform the desired operations and return the results to VLISP in a variant form. You can develop wrapper DLL’s for almost any exposed service using Visual Basic and invoke that DLL from Visual LISP or VBA and do whatever you need to do.

Chapter 22 –Using Visual Basic DLLs with Visual LISP

Now that you've seen how to use interfaces to other applications, it's time to consider making your own custom tools. More accurately, this involves developing your own services as components that can be referenced by Visual LISP (or other ActiveX language options like VB or VBA). For example, you can develop your own ActiveX controls or DLLs and use them from Visual LISP. This opens up an unlimited potential for creating efficient pathways to other resources from within Visual LISP.

One such example might be to define a set of database interface routines that execute stored procedures and returns them as a list to your Visual LISP application. Then the DLL can take care of the ADO connections and doing the commands and recordset management itself. This frees you from having to worry about doing this in Visual LISP, which although can be done, it is much more tedious to do than with more suitable languages like Visual Basic or Delphi. Conversely, the use of a VLX application function library allows you to provide a similar purpose for other LISP or Visual LISP applications. Huh? Yes, you can wrap DLL functionality within VLX applications so that even your calls to your DLL's remain private and protected.

As an example, we'll create an ActiveX DLL that performs a simple function of concatenating strings and returning a combined string result. This will involve using Microsoft Visual Basic 6.0 and a new ActiveX DLL project (see Figure 22-1 below).

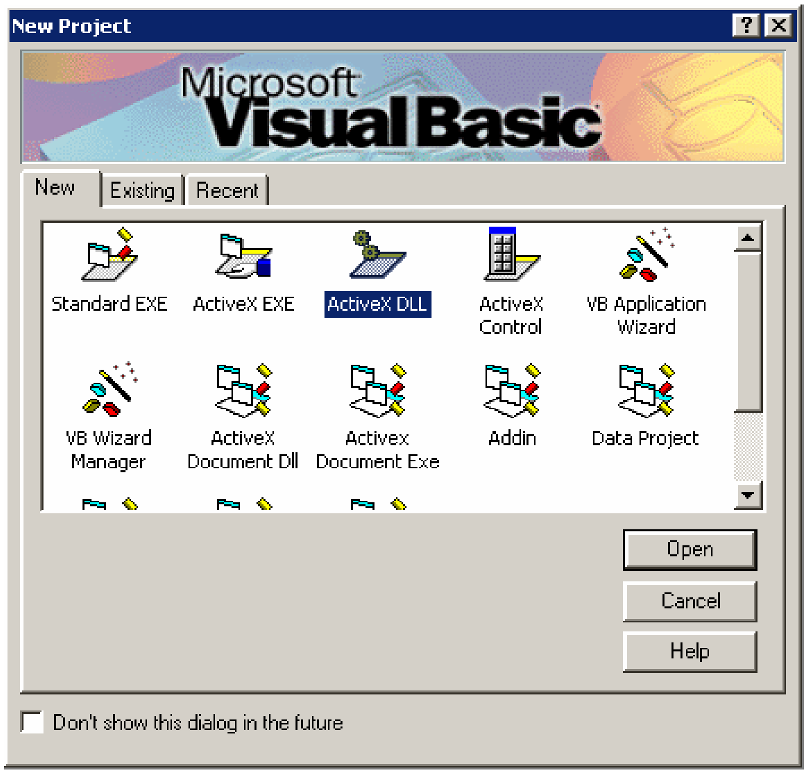
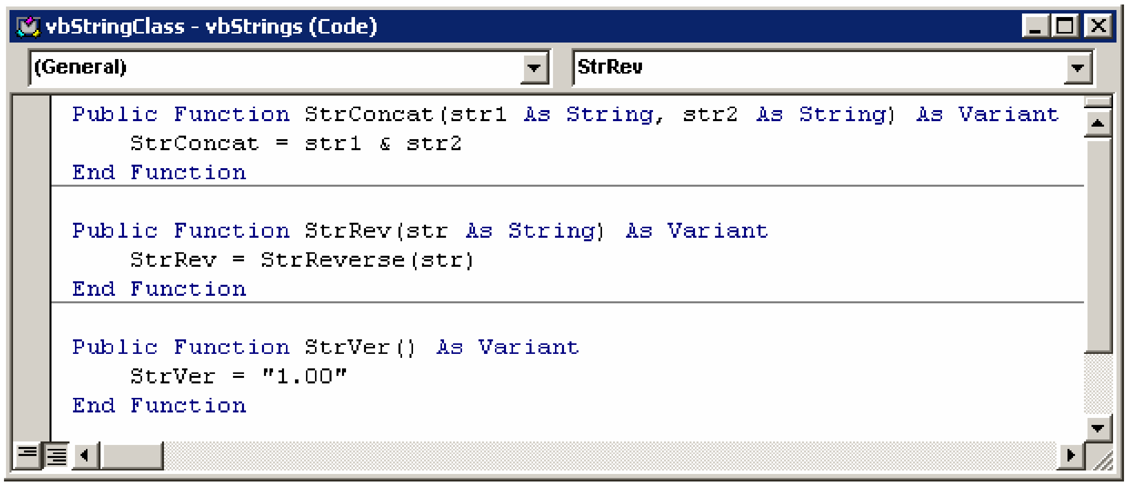


Figure 22-1 – The Microsoft Visual Basic 6.0 *New Project* form.

Once you pick the Open button, the Visual Basic 6.0 development environment will open and a default code window will be displayed. Change the name of the default project from Project1 to **vbStringClass**, and change the name of the default class module from Class1 to **vbStrings**. Then, enter the code shown in Figure 22-2 in the code window to define three distinct Public Functions. A *public function* is one that can be exposed to any ActiveX consumer when the vbStringClass class is loaded.



```
Public Function StrConcat(str1 As String, str2 As String) As Variant
    StrConcat = str1 & str2
End Function

Public Function StrRev(str As String) As Variant
    StrRev = StrReverse(str)
End Function

Public Function StrVer() As Variant
    StrVer = "1.00"
End Function
```

Figure 22-2 – Creating a public function in the class module code window.

Once you have entered the code to define the class function, save the class module as vbStrCat.cls and the project itself as vbStringClass.vbp. Then pick the File pulldown menu and select the *Make vbStringClass.DLL* option. Pick the OK button on the form that appears and Visual Basic will compile your class module into a DLL and register it on the local operating system. This DLL is now ready for use by any other program, be it Visual LISP, Visual Basic, C/C++, Delphi, ASP or whatever. The next step is to load this DLL using its TypeLib interface within Visual LISP and try it out.

Tip! You should avoid using the variable/symbol name "acad" in your program code. Some third-party VLX applications will apply symbol protection to this name and it may cause you to experience an error message when you try to use that name in your code.

Open the Visual LISP editor, create a new code window and enter the following code, with these three distinct LISP functions.

```
(vl-load-com)
(defun vbStrCat (string1 string2 / $acad vbstrcls out)
  (setq $acad (vlax-get-acad-object))
  (setq vbstrcls
    (vla-GetInterfaceObject $acad "vbStringClass.vbStrings")
  )
  (setq out (vlax-invoke-method vbstrcls "StrConcat" string1 string2))
  (vlax-release-object vbstrcls)
  (vlax-release-object $acad)
  out
)
```

```
(defun vbStrRev (string / $acad vbstrcls out)
  (setq $acad (vlax-get-acad-object))
  (setq vbstrcls
    (vla-GetInterfaceObject $acad "vbStringClass.vbStrings")
  )
  (setq out (vlax-invoke-method vbstrcls "StrReverse" string))
  (vlax-release-object vbstrcls)
  (vlax-release-object $acad)
  out
)
```

```
(defun vbStrVer ( / $acad vbstrcls out)
  (setq $acad (vlax-get-acad-object))
  (setq vbstrcls (vla-GetInterfaceObject $acad
"vbStringClass.vbStrings"))
  (setq out (vlax-invoke-method vbstrcls "StrVer"))
  (vlax-release-object vbstrcls)
  (vlax-release-object $acad)
  out
)
```

Figure 22-3 – Visual LISP code to implement the DLL class functions.

Each `(defun)` function uses the AcadApplication object method "GetInterfaceObject" to fetch your registered DLL from the operating system and expose the class functions within your Visual LISP code. Notice how the objects are explicitly released before returning result values.

Tip!

When working with Visual Basic, it is important to be careful about defining *Functions* as opposed to *Sub* routines. Functions can return values, Subs cannot. Also, if you fail to use the Function=Result return at the end of a given function, the return will be *nil* to your LISP expressions. You might expect an ActiveX error to be generated, but this is not the case. Also, you must define the return data type as a Variant for all Functions in order to use them with VLISP. If you retrun some other data type, it will cause an ActiveX error in your LISP code because it expects a Variant data type.

Now, load this Visual LISP code into AutoCAD and test it by entering these function examples following at the AutoCAD command prompt:

```
Command: (vbStrCat "THE " "DOG")
```

This should return something like the following result:

```
Initializing VBA System… "THE DOG"
Command: (vbStrRev "THE DOG BARKED") returns "DEKRAB GOD EHT"
Command: (vbStrVer) returns "1.00"
```

This is a very simple example and is only intended to demonstrate that you can develop components in other language tools and use them from Visual LISP and other language environments to get the job done. Note that the first time you reference an imported function like this, you will see a notice saying "Initializing VBA System…" just before the result is returned. This is because AutoCAD uses the VBA system services to interact with ActiveX DLL components that involve certain ActiveX functions. After the first invocation in a given drawing session, you will not see that message again, only the return value.

Registering DLLs

If you plan to use this approach, you need to be aware of how DLL components are used and how to *register* them on a given computer. When you compile an ActiveX DLL in Visual Basic, it handles this chore for you on your local machine. But other users on other machines will have to register the DLL another way before it can be used on their machine. The Windows REGSVR32 command is used to manually register DLL components a given machine. The syntax is REGSVR32 filename.dll where filename.dll is the full path and filename to where the DLL resides (it can be local or on a network share).

For example, if you were to deploy this DLL we created above on another machine, and you had copied the vbStringClass.DLL file to a network folder named G:\acadsupport\components\vbStringClass.DLL you would use the following command on each client computer:

```
C:\>REGSVR32 G:\ACADSUPPORT\COMPONENTS\VBSTRINGCLASS.DLL
```

This can be done from any command prompt (on that local machine), or through a SHELL operation from within AutoCAD (again, on that local machine), or programmatically within Visual LISP using some code to check for the registration and taking care of registering the DLL if it hasn't already been registered on the machine.

Re-Registering DLLs

Whenever you release an updated version of your custom DLL it will also receive a new GUID identifier. This lets other applications know what specific version of your DLL they are using. In order to update a DLL on another machine, you must first un-register it using REGSVR32 /U and then re-register it using REGSVR32 to install the newer version and register the new GUID. The function shown in figure 14-1 `(dllunregister)` can be used from within VLISP to unregister a known DLL. Then you can use the `(dllregister)` function to register a newer version.

This type of approach is very common and forms the basis of how many web development environments work. One example is with ASP web programming and the use of MTS (Microsoft Transaction Server) with Visual Basic DLLs. The ASP code can invoke the DLL just as we can do with Visual LISP, in order to hand off complex processing tasks to a dedicated component, which in turn returns the result to ASP for use within rendering a web page to the user. DLL libraries provide their functionality to all ActiveX languages at once. So when you create a new DLL, remember that it can normally be used by VLISP, VBA, VB, C/C++, C#, Delphi, Java, ActivePerl, WSH and many more languages.

Finally, if you do decide to pursue creating custom DLL components, you should seriously consider using the Class Builder Wizard add-in for Visual Basic 6.0. This handy utility helps you define a new class and develop an object model for your own DLL classes. This opens up yet greater possibilities to you as well as other programmers using other ActiveX-enabled languages in your organization.

Tip!

Now, would be a good time to start taking your VB programming buddies out to lunch and buying them a beer or two. While it's cool to learn other programming languages, it's even cooler to form partnerships or teams where each member can contribute a unique skill set to help solve complex tasks in creative ways. Most experienced VB programmers, like programmers of all language tools, have built libraries of powerful gadgets that could unlock enormous potential for you as an AutoCAD developer. While you're out scouting for talent, don't forget the folks the work with other powerful tools like Delphi, C++, Scripting, SQL, Java, XML, and even web developers.

Chapter 23 – Working With Dialog Forms

Since Visual LISP hasn't added anything new to the world of DCL interfacing from LISP, this chapter will instead focus on how DCL can be used within VLX applications. There are quite a few methods for dealing with callbacks from dialog forms. I am not going to preach any particular method to you. The methods shown herein are only my own habitual ways of working with DCL callbacks. If you have other preferences, please continue on your merry way unless you feel like changing habits now.

Referencing DCL Definitions

In the old days of AutoLISP, you had two files for any application that used a dialog form. You had the LSP file and the DCL file, and both had to be present and available to the user at runtime to execute the application successfully. With the new VLISP Application features, you can now compile the LSP and DCL source code files into a single VLX application file to provide to your users. This not only protects your source code somewhat, but it makes for easier deployment and maintenance, especially in a networked or distributed environment.

For example, in AutoLISP you might setup a LSP application to access the DCL source as follows (again, your style may be different. This is for example only):

```
(cond
  ( (setq dcfl (findfile “mydialog.dcl”))
    (setq dcid (load_dialog dcfl))
    (cond
      ( (new_dialog “myform” dcid)
        ...do something with dialog callbacks here...
        (action_tile “accept” “(done_dialog 1)”)
        (action_tile “cancel” “(done_dialog 0)”)
        (start_dialog dlgstatus)
        (unload_dialog dcid)
      )
    )
  )
  ( T (princ “\nUnable to locate DCL form file!”) )
)
```

You can still use this “legacy” code with only very minor changes to enable it to be compiled into a Visual LISP VLX application file. Simply remove the check for the file location and assume it is always there (because, when you compile it into the VLX it will be). You would only need to check for the form loading itself, which is already being done in the original code.

```
(setq dcid (load_dialog “mydialog”))
(cond
  ( (new_dialog “myform” dcid)
    ...leave the rest as-is...
  )
)
```

As you can see, it also shortens your code a bit. You can use this to help guide you in porting your older AutoLISP dialog form applications into Visual LISP VLX applications by referring to chapter 13 for how to build VLX applications.

Dynamic Dialog Interaction

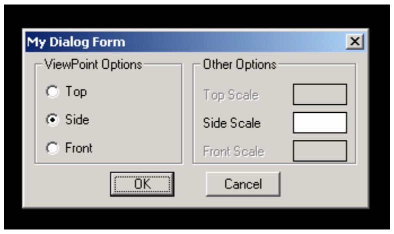
A very common feature requirement in dialog-based applications is the need to have form features dynamically change in response to user interaction. Such things as changing an image or edit box values based upon selections in other parts of the form. Probably the most common is the need to enable or disable features based upon user selections.

The following example will step through how to make a dialog form that enables or disables certain features based upon selections in another part of the form. First, we'll show the example dialog form definition:

```
// save as MyDialog.dcl

myform : dialog {
  label = “My Dialog Form”;
  : row {
    : boxed_radio_column {
      key = “viewpoint”;
      label = “ViewPoint Options”;
      : radio_button {key = “TOP”; label = “Top”; }
      : radio_button {key = “SIDE”; label = “Side”; }
      : radio_button {key = “FRONT”; label = “Front”; }
    }
    : boxed_column {
      label = “Other Options”;
      : edit_box {key = “TOP2”; label = “Top Scale”; edit_width = 6; }
      : edit_box {key = “SIDE2”; label = “Side Scale”; edit_width = 6; }
      : edit_box {key = “FRONT2”; label = “Front Scale”; edit_width = 6; }
    }
  }
  ok_cancel;
}
```

Now, we'll see how to enable only one of the edit-boxes at a time, with respect to which of the radio-buttons are selected on the left-hand side of the form. To do this, we'll define a few functions. The first will handle the dialog form and the call-backs using an `(action_tile)` call-back for the radio_column “viewpoint”, which will receive the key-name of the radio button selection within its collection. Then we can use that key name to perform a conditional action using the other two functions to manipulate the other tiles.

```
;;; Save as MyDialog.lsp

(defun C:Myform ( / *error* dcid ok choice)
  (defun *error* (s)
    (princ (strcat "\nError: " s))
    (vl-bt)
    (princ)
  )
  (setq dcid (load_dialog "mydialog"))
  (if (null $MYFORM1) (setq $MYFORM1 "TOP"))
  (cond
    ( (new_dialog "myform" dcid); form name is case sensitive!
      (set_tile "viewpoint" $MYFORM1)
      (change-form $MYFORM1); initialize tiles using default value
      (action_tile "viewpoint" "(setq choice (change-form $value))")
      (action_tile "accept" "(setq ok 1)(done_dialog)")
      (action_tile "cancel" "(setq ok 0)(done_dialog)")
      (start_dialog)
      (unload_dialog dcid)
      (if (= ok 1)
        (if choice (setq $MYFORM1 choice))
      )
    )
    ( T (princ "\nUnable to load form from dialog definition.") )
  )
  (princ)
)

;;; apply tile mode changes in response to given tile to enable

(defun change-form (val)
  (cond
    ( (= val "TOP")
      (mode-tiles '("TOP2") 0)
      (mode-tiles '("SIDE2" "FRONT2") 1)
    )
    ( (= val "SIDE")
      (mode-tiles '("SIDE2") 0)
      (mode-tiles '("TOP2" "FRONT2") 1)
    )
    ( (= val "FRONT")
      (mode-tiles '("FRONT2") 0)
      (mode-tiles '("TOP2" "SIDE2") 1)
    )
  )
  val
)

;;; apply tile mode to list of tilenames

(defun mode-tiles (tiles mode)
  (foreach tile tiles (mode_tile tile mode))
)
```



Now simply follow the steps in Chapter 13 to compile both of these source files into a single VLX application file and load it into your AutoCAD session. Type in MYFORM to run the command and see how the edit boxes react when you select the radio buttons.

`Your dialog display should look something like this example image if you compile and load it correctly.`

Controlling Images From Call-Backs

Now that we’ve seen how to enable and disable tiles from call-back values, let’s try going a tiny step farther and use this to change the slide image in a dialog form image tile. To demonstrate this, you should load the sample code slide files into a common folder and make sure the folder is in your default search path. Once we compile the VLX application and load it, it will still need to find the .SLD files used for the image tile display. Unfortunately, VLISP doesn't provide a means to compile slide files into the VLX as it does for DCL, DVB, INI and LSP files.

Let's take the dialog form defined above and modify it slightly to add a single image_tile:

```
// save as MyDialog2.dcl

myform : dialog {
  label = "My Dialog Form";
  : row {
    : boxed_column {
      label = "Preview";
      : image_button {
        key = "image";
        height = 10;
        aspect_ratio = 1.25;
        color = 0;
        fixed_height = true;
        fixed_width = true;
      }
    }
    : boxed_radio_column {
      key = "viewpoint";
      label = "ViewPoint Options";
      : radio_button {key = "TOP"; label = "Top"; }
      : radio_button {key = "SIDE"; label = "Side"; }
      : radio_button {key = "FRONT"; label = "Front"; }
    }
    : boxed_column {
      label = "Other Options";
      : edit_box {key = "TOP2"; label = "Top Scale"; edit_width = 6; }
      : edit_box {key = "SIDE2"; label = "Side Scale"; edit_width = 6; }
      : edit_box {key = "FRONT2"; label = "Front Scale"; edit_width = 6; }
    }
  }
  ok_cancel;
}
```

In order to properly control the image_tile, we should define a special function that takes care of finding the image file and adjusting scaling to suit the DCL configuration. You can load this example from MyDialog2.LSP in the sample files collection for this book. If you look closely, you'll notice that this function accepts either a SLD or SLB (slide-library) file, thereby making it possible to bundle your slides into a SLB and keep the total deployment down to just the VLX and SLB files.

```
(defun slide-show
  (tile sld slb / w ky xc yc sldnam)
  (cond
    ( (or
        (and slb (findfile slb))
        (findfile sld)
      )
      (setq
          xc  (dimx_tile tile)
          yc  (dimy_tile tile)
      )
      (start_image tile)
      (fill_image 0 0 xc yc 0)
      (if slb
        (progn
          (setq slb (vl-filename-base slb))
          (setq sldnam (strcat slb "(" sld ")" ))
        )
        (setq sldnam sld)
      )
      (slide_image 0 0 xc yc sldnam)
      (end_image)
    )
    ( T (alert "Slide image file not found...") )
  )
)
```

Assuming that you use the provided slide files TOP.SLD, SIDE.SLD and FRONT.SLD and place them in a folder that is in the current default search path, you should be able to compile and load the MYDIALOG2.VLX and run it successfully.

Chapter 24 – Examples of Common Tasks

This chapter will suggest some ideas for solving common tasks that combine aspects of previous chapters. You may find some of these useful, maybe not. In any case, they are provided simply to demonstrate how VLISP can be used to do things AutoLISP alone is not capable of doing.

Example 1 – Dumping a List of Layer Properties

This example involves the task of producing an HTML report of all layers in the current drawing, including their properties (color, linetype, etc.) and opening the report in a web browser after completion. When loaded, the command is DUMPLAYERS.

```
(defun C:DUMPLAYERS
  ( / acad doc dwg layers name col ltp lwt pst onoff frz dat
      path olist outfile output)
  (vl-load-com)
  (setq acad   (vlax-get-acad-object)
        doc    (vla-get-activedocument acad)
        dwg    (vla-get-name doc)
        path   (vla-get-path doc)
        layers (vla-get-layers doc)
  )
  (vlax-for each layers
    (setq name  (vla-get-name each)
          col   (itoa (dsx-get-color each));; see Chapter 25!
          ltp   (vla-get-linetype each)
          lwt   (itoa (vla-get-lineweight each))
          pst   (vla-get-plotstylename each)
          onoff (if (= :vlax-true (vla-get-layeron each))
                  "ON" "OFF"
                )
          frz   (if (= :vlax-true (vla-get-freeze each))
                  "FROZEN" "THAWED"
                )
          dat   (list name col ltp lwt pst onoff frz)
          olist (cons dat olist)
    )
  ); vlax-for

  (vlax-release-object layers)
  (vlax-release-object doc)
  (vlax-release-object acad)

  (cond
    ( olist
      (setq outfile (strcat (vl-filename-base dwg) ".htm"))
      (setq outfile (strcat path outfile))
      (cond
        ( (setq output (open outfile "w"))
          (write-line "<html>" output)
          (write-line "<head><title>" output)
          (write-line (strcat "Layer Dump: " dwg) output)
          (write-line "</title></head><body>" output)
          (write-line (strcat "<b>Drawing: " dwg "</b><br>") output)
          (write-line "<table border=1>" output)
          (foreach layset olist
            (write-line "<tr>" output)
            (foreach prop layset
              (write-line (strcat "<td>" prop "</td>") output)
            )
            (write-line "</tr>" output)
          ); foreach layer set
          (write-line "</table></body></html>" output)
          (close output)
          (setq output nil)
          (princ "\nReport finished! Opening in browser...")
          (vl-cmdf "_.browser" outfile)
        )
        ( T (princ "\nUnable to open output file.") )
      )
    )
    ( T (princ "\nUnable to get layer table information.") )
  )
)
```

Example 2 – Set All Entities to "ByLayer"

This example involves the task of assigning all entities in the current working space to "ByLayer" with respect to Color, Linetype, and Lineweight properties.

```
(defun C:BYLAYER
  ( / acad doc ssall i obj)
  (vl-load-com)
  (setq acad (vlax-get-acad-object)
        doc  (vla-get-activedocument acad)
  )
  (vla-startundomark acad)
  (vla-zoomextents acad)
  (cond
    ( (setq ssall (ssget "x")); get all entities
      (setq i 0)
      (repeat (sslength ssall)
        (setq obj (vlax-ename->vla-object (ssname ssall i)))
        (dsx-put-color obj acByLayer);; See Chapter 25!
        (vlax-put-property obj "Linetype" "ByLayer")
        (vlax-put-property obj "Lineweight" acLnWtByLwDefault)
        (vlax-release-object obj)
        (setq i (1+ i))
      )
    )
  )
  (vla-endundomark acad)
  (princ "\nFinished processing all entities.")
  (princ)
)
```

Example 3 – Purge, Audit and Save all Opened Drawings

This example involves the task of iterating through the Documents collection and performing a Purge, Audit and Save operation on each document.

```
(defun C:DOALL ( / $acad docs dnum this)
  (vl-load-com)
  (setq $acad (vlax-get-acad-object)
        docs (vla-get-documents $acad)
        this (vla-get-activedocument $acad)
        dnum (vla-get-count docs)
  )
  (vlax-for each docs
    (vla-purgeall each)
    (vla-auditinfo each T)
    (vla-save each)
  )
  (vla-get-activedocument this)
  (vlax-release-object docs)
  (vlax-release-object this)
  (vlax-release-object $acad)
  (princ (strcat “\nProcessed “ (itoa dnum) “ drawings.”))
  (princ)
)
```

Example 4 – Zoom Extents and Save all Opened Drawings

This example involves iterating through each opened document, zooming to extents in the current (active) space and tab, and saving the document. Finally, it returns to the starting document when finished.

```
(defun C:ZOOMALL ( / $acad docs dnum this)
  (vl-load-com)
  (setq $acad (vlax-get-acad-object)
        docs  (vla-get-documents $acad)
        this  (vla-get-activedocument $acad)
        dnum  (vla-get-count docs)
  )
  (vlax-for each docs
    (vla-put-ActiveDocument each)
    (vla-ZoomExtents $acad)
    (vla-save each)
  )
  (vla-put-activedocument this)
  (vlax-release-object docs)
  (vlax-release-object this)
  (vlax-release-object $acad)
  (princ (strcat “\nProcessed “ (itoa dnum) “ drawings.”))
  (princ)
)
```

Chapter 25 - Changes in AutoCAD 2004

I'm not going to provide an exhaustive review of every difference introduced between AutoCAD 2002 and 2004. I'll leave that for Autodesk and other people to provide. The following section describes some of the changes since AutoCAD 2002 with respect to the ActiveX interface to AcadPreferences collections, as well as the System Variables collection. Items shown in blue are new since AutoCAD 2002. Items shown in red are modified since AutoCAD 2002.

General Changes

The most significant general programming change is the conformity to Microsoft guidelines. In particular: Windows XP compliance. The result is a completely revised path and registry scheme that embraces the recommendations of Microsoft. Users no longer must be members of the local *Administrators*, or *Power Users* group to launch AutoCAD. Nor do they require hacked regsistry permissions or policy templates to open "holes" into the registry for restricted user access. The trade-off is added complexity, not much, but particularly under the user profile folder tree. This will become apparent as you examine the new folder paths under the AcadPreferencesFiles collection (see below).

ObjectARX (and ObjectDBX as well) have been updated and recompiled with Microsoft Visual C++ 7.0 (part of Visual Studio .NET). The result is that ARX and DBX components built for use on versions prior to AutoCAD 2004 will no longer work. If you have any such works, you will need to recompile them on VC++ 7.0 for use with AutoCAD 2004. Case in point is DOSlib by Robert McNeel Inc. (www.mcneel.com) which has released version 6.03 specifically for use with AutoCAD 2004.

Type Libraries must now be invoked and addressed using their *version-specific* GUID names. In other words, where you might have invoked AutoCAD as `AutoCAD.Application`, you now must use `AutoCAD.Application.16`.

Visual LISP Changes

Technically speaking, nothing has changed *functionally* with respect to Visual LISP. However, there are a few new oddities. The VLIDE.DSK file which stores your IDE configuration settings is now stored under your Windows Profile folder (on Windows 2000 and XP systems: \Documents and Settings\<username>\Local Settings\Application Data\Autodesk\AutoCAD\<language-code>\VLIDE.DSK).

The (vlax-get) function works fine, however the (vlax-put) function is unpredictable. In some cases it does nothing (does not apply a new value to a specified property). The function (vlax-invoke) works reliably from my tests. For VLISP programming in AutoCAD 2004, you should make a habit of using the full property and method inferences to avoid problems. For example, if you've been using (vlax-get object propertyname), you should instead use (vla-get-propertyname object), or even (vlax-get-property object propertyname).

True Color Properties

As for VLISP (or other) coding changes, the addition of “true color” properties requires an adjustment to some code. This actually turned out to be the only significant coding changes I was forced to address. Once you get familiar with RGB color mapping this won’t be a major concern for you.

Here is an example function you can use to hide the mess from the rest of your code. Instead of using (vla-put-color) and (vla-get-color), you can use custom functions like these. I’m cheating a bit here by using the DOSlib 6 (dos_acitorbg) function, but this is ony one possible solution. Thanks for Jon Szewczak for this example:

```
(defun dsx-put-color (obj num / av)
  (setq av (substr (getvar "acadver") 1 2))
  (if (>= av "16")
    ;; if AutoCAD 2004...
    (dsx-put-color2004 obj num)
    ;; if any other version...
    (dsx-put-property obj "Color" num)
  )
)

(defun dsx-put-color2004 (obj num / oColor numlst)
  (if
    (not
      (vl-catch-all-error-p
        (setq oColor
          (vl-catch-all-apply 'vla-get-TrueColor (list obj))
        )
      )
    )
    (progn ;; if getting the TrueColor object of 'obj' did not return a vla-error
      (cond
        ( (= 'INT (type num)) ;; if an ACI index integer is passed
          ;; if obj is a table record (i.e. Layer)...
          (if (vl-string-search "Table" (vla-get-ObjectName obj))
            (progn
              (vla-put-ColorMethod oColor acColorMethodByACI)
              (vla-put-ColorIndex oColor num)
            )
            (if (= num acBylayer) ;; if obj is an entity
              (progn ;; if num is to be byLayer
                (vla-put-ColorMethod oColor acColorMethodByLayer)
                (vla-put-ColorIndex oColor acByLayer)
              )
              (progn ;; if num is to be an override
                (vla-put-ColorMethod oColor acColorMethodByACI)
                (vla-put-ColorIndex oColor num)
              )
            )
          )
        )
        ( (and (listp num) (= (length num) 3)) ;; an RGB list is passed
          (vla-put-ColorMethod oColor acColorMethodByRGB) ;; set the method
          ;; set the RGB values
          (vlax-invoke-method oColor 'SetRGB (nth 0 num) (nth 1 num) (nth 2 num))
        )
      )
      ;; stuff color object back into parent object
      (vla-put-TrueColor obj oColor)
      ;; clean up the memory stack of unused objects
      (vlax-release-object oColor)
    )
    (vl-catch-all-error-message oColor)
  )
)

(defun dsx-put-property (obj prop val / try)
  (cond
    ( (and
        (vlax-property-available-p obj prop)
        (not
          (vl-catch-all-error-p
            (setq try
              (vl-catch-all-apply 'vlax-put-property (list obj prop val))
            )
          )
        )
      )
      val
    )
  )
)

(defun dsx-get-property (obj prop / try)
  (cond
    ( (and
        (vlax-property-available-p obj prop)
        (not
          (vl-catch-all-error-p
            (setq try (vl-catch-all-apply 'vlax-get (list obj prop)))
          )
        )
      )
      try
    )
  )
)

;;; Provides a concatenated string result of the RGB color list using
;;; comma delimiters.  For example, if the RGB list is (255 200 155), the
;;; return value would be "255,200,155"

(defun dsx-acitorgb (cnum / cmap)
  (setq cmap (dos_acitorgb cnum))
  (strcat
    (nth 0 cmap) "," (nth 1 cmap) "," (nth 2 cmap)
  )
)
```

Important: It is worth noting that Color-Based Plot Styles (CTB) do not work with drawings that contain true color mapping to entitiess or layer tables. The result is that the plot style is simply ignored, both during preview and final output. This could possibly affect code you've written to automate plotting via the ActiveX interfaces. For example, if you assign color 255,148,228 to a given layer and attempt to use monochrome.ctb, the result will still be a color print (verify using Preview beforehand).

Changes to the ObjectDBX Interface

Due to the updated version of ObjectDBX in AutoCAD 2004, you need to replace your references to the DLL from "AxDb15.dll" to "AxDb16.dll". As mentioned in the beginning section of this chapter, the GUID string identifier must be updated from "ObjectDBX.AxDbDocument" to "ObjectDBX.AxDbDocument.16" to denote the class version. This is actually a Microsoft guideline and was adopted during the porting of AutoCAD 2004 to using Microsoft Visual C++ 7.0 (part of Visual Studio.NET).

Changes to External Referencing of AcadApplication

As with ObjectDBX, you need to update your references to the AutoCAD Application object when invoking it from an external source. This doesn't really affect Visual LISP since it works from within AutoCAD, but it's good to know nonetheless. The new GUID string name is "AutoCAD.Application.16". If you're developing with Microsoft Visual Studio 6 or .NET, or Borland Delphi or some other "visual" development product, this is not really an issue as it is usually apparent while setting a project reference to the required interface library.

Changes to AcadPreferences

The Preferences collections have been revised somewhat to suit some of the new changes in AutoCAD 2004. Among these are pathing, display, and a few others. It is worth noting that the Profiles collection has not been modified.

For the sake of clarity and improved printing I have substituted <%apppath%> for the new default user profile path used by AutoCAD 2004 on Windows 2000 and XP operating systems: `"%userprofile%\Application Data\Autodesk\AutoCAD 2004\R16.0\enu"` and I've substituted `<%localpath%>` for `"%userprofile%\Local Settings\Application Data\Autodesk\AutoCAD 2004\R16.0\enu"`

```
; IAcadPreferences:
; This object specifies the current AutoCAD settings
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   Display (RO) = #<VLA-OBJECT IAcadPreferencesDisplay 037faa6c>
;   Drafting (RO) = #<VLA-OBJECT IAcadPreferencesDrafting 037faa68>
;   Files (RO) = #<VLA-OBJECT IAcadPreferencesFiles 037faa70>
;   OpenSave (RO) = #<VLA-OBJECT IAcadPreferencesOpenSave 037faa74>
;   Output (RO) = #<VLA-OBJECT IAcadPreferencesOutput 037faa78>
;   Profiles (RO) = #<VLA-OBJECT IAcadPreferencesProfiles 037faa7c>
;   Selection (RO) = #<VLA-OBJECT IAcadPreferencesSelection 037faa80>
;   System (RO) = #<VLA-OBJECT IAcadPreferencesSystem 037faa84>
;   User (RO) = #<VLA-OBJECT IAcadPreferencesUser 037faa88>
; No methods

; IAcadPreferencesDisplay:
; This object contains the options from the Display tab on the Options dialog
;
; Note: DockedVisibleLines is ignored
;
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   AutoTrackingVecColor = unsupported result type: 19
;   CursorSize = 5
;   DisplayLayoutTabs = -1
;   DisplayScreenMenu = 0
;   DisplayScrollBars = -1
;   DockedVisibleLines = 3
;   GraphicsWinLayoutBackgrndColor = unsupported result type: 19
;   GraphicsWinModelBackgrndColor = unsupported result type: 19
;   HistoryLines = 400
;   ImageFrameHighlight = 0
;   LayoutCreateViewport = 0
;   LayoutCrosshairColor = unsupported result type: 19
;   LayoutDisplayMargins = -1
;   LayoutDisplayPaper = 0
;   LayoutDisplayPaperShadow = 0
;   LayoutShowPlotSetup = 0
;   MaxAutoCADWindow = 0
;   ModelCrosshairColor = unsupported result type: 19
;   ShowRasterImage = 0
;   TextFont = "Courier"
;   TextFontSize = 10
;   TextFontStyle = 0
;   TextWinBackgrndColor = unsupported result type: 19
;   TextWinTextColor = unsupported result type: 19
;   TrueColorImages = -1
;   XRefFadeIntensity = 50
; No methods

; IAcadPreferencesDrafting:
; This object contains the options from the Drafting tab on the Options dialog
; Property values:
;   AlignmentPointAcquisition = 0
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   AutoSnapAperture = 0
;   AutoSnapApertureSize = 10
;   AutoSnapMagnet = -1
;   AutoSnapMarker = -1
;   AutoSnapMarkerColor = 2
;   AutoSnapMarkerSize = 5
;   AutoSnapTooltip = -1
;   AutoTrackTooltip = -1
;   FullScreenTrackingVector = -1
;   PolarTrackingVector = -1
; No methods

; IAcadPreferencesFiles:
; This object contains the options from the Files tab on the Options dialog
; Property values:
;   AltFontFile = "simplex.shx"
;   AltTabletMenuFile = ""
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   AutoSavePath = "<%userprofile%>\\Local Settings\\Temp\\"
;   ColorBookPath = "C:\\Program Files\\AutoCAD
2004\\support\\color; <%apppath%>\\support\\color"
;   ConfigFile (RO) = "<%localpath%>\\acad2004.cfg"
;   CustomDictionary = "<%apppath%>\\support\\sample.cus"
;   DefaultInternetURL = "http://www.autodesk.com"
;   DriversPath = "C:\\Program Files\\AutoCAD 2004\\drv"
;   FontFileMap = "<%profilepath%>\\support\\acad.fmp"
;   HelpFilePath = "C:\\Program Files\\AutoCAD 2004\\help\\acad.chm"
;   LogFilePath = "<%apppath%>\\"
;   MainDictionary = "enu"
;   MenuFile = "<%apppath%>\\support\\acad"
;   PostScriptPrologFile = ""
;   PrinterConfigPath = "<%apppath%>\\plotters"
;   PrinterDescPath = "<%apppath%>\\plotters\\PMP Files"
;   PrinterStyleSheetPath = "<%apppath%>\\Plot Styles"
;   PrintFile = "."
;   PrintSpoolerPath = "<%userprofile%>\\Local Settings\\Temp\\"
;   PrintSpoolExecutable = ""
;   SupportPath = "<%apppath%>\\support;C:\\Program Files\\AutoCAD
2004\\support;C:\\Program Files\\AutoCAD 2004\\fonts;C:\\Program Files\\AutoCAD
2004\\help;C:\\Program Files\\AutoCAD 2004\\express;C:\\Program Files\\AutoCAD
2004\\support\\color"
;   TempFilePath = "<%userprofile%>\\Local Settings\\Temp\\"
;   TemplateDwgPath = "<%localpath%>\\Template"
;   TempXrefPath = "<%userprofile%>\\LOCALS~1\\Temp\\"
;   TextEditor = "Internal"
;   TextureMapPath = "<%localpath%>\\textures"
;   ToolPalettePath = "<%apppath%>\\support\\ToolPalette"
;   WorkspacePath = "<%apppath%>\\Data Links"
; Methods supported:
;   GetProjectFilePath (1)
;   SetProjectFilePath (2)

; IAcadPreferencesOpenSave:
; This object contains the options from the Open and Save tab on the
; Options dialog
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   AutoAudit = 0
;   AutoSaveInterval = 10
;   CreateBackup = -1
;   DemandLoadARXApp = 3
;   FullCRCValidation = 0
;   IncrementalSavePercent = 50
;   LogFileOn = 0
;   MRUNumber (RO) = 9
;   ProxyImage = 1
;   SaveAsType = 24 (24=Acad 2004, 12=Acad 2000-2002)
;   SavePreviewThumbnail = -1
;   ShowProxyDialogBox = -1
;   TempFileExtension = "ac$"
;   XrefDemandLoad = 2
; No methods

; IAcadPreferencesOutput:
; This object contains the options from the Output tab on the Options dialog
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   DefaultOutputDevice = "<%apppath%>\\plotters\\PlotterDevice1.pc3"
;   DefaultPlotStyleForLayer = "ByColor"
;   DefaultPlotStyleForObjects = "ByColor"
;   DefaultPlotStyleTable = "monochrome.ctb"
;   OLEQuality = 1
;   PlotLegacy = 0
;   PlotPolicy = 1
;   PrinterPaperSizeAlert = -1
;   PrinterSpoolAlert = 0
;   UseLastPlotSettings = 0
; No methods

; IAcadPreferencesProfiles:
; This object contains the options from the Profiles tab on the Options dialog
; Property values:
;   ActiveProfile = "default"
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
; Methods supported:
;   CopyProfile (2)
;   DeleteProfile (1)
;   ExportProfile (2)
;   GetAllProfileNames (1)
;   ImportProfile (3)
;   RenameProfile (2)
;   ResetProfile (1)

; IAcadPreferencesSelection:
; This object contains the options from the Selection tab on the Options dialog
```

```
;
; Note: There is no property for hover grip color, use GRIPHOVER sysvar
;
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   DisplayGrips = -1
;   DisplayGripsWithinBlocks = 0
;   GripColorSelected = 1
;   GripColorUnselected = 160
;   GripSize = 5
;   PickAdd = -1
;   PickAuto = -1
;   PickBoxSize = 3
;   PickDrag = 0
;   PickFirst = -1
;   PickGroup = -1
; No methods

; IAcadPreferencesSystem:
; This object contains the options from the System tab on the Options dialog
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   BeepOnError = 0
;   DisplayOLEScale = -1
;   EnableStartupDialog = -1
;   LoadAcadLspInAllDocuments = 0
;   ShowWarningMessages = -1
;   SingleDocumentMode = 0
;   StoreSQLIndex = -1
;   TablesReadOnly = 0
; No methods

; IAcadPreferencesUser:
; This object contains the options from the User tab on the Options dialog
; Property values:
;   ADCInsertUnitsDefaultSource = 1
;   ADCInsertUnitsDefaultTarget = 1
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00af9594>
;   HyperlinkDisplayCursor = -1
;   HyperlinkDisplayTooltip = -1
;   KeyboardAccelerator = 1
;   KeyboardPriority = 2
;   SCMCommandMode = 1
;   SCMDefaultMode = 1
;   SCMEditMode = 1
;   SCMTimeMode = -1
;   SCMTimeValue = 250
;   ShortCutMenuDisplay = -1
; No methods

; The SCMTimeMode value controls whether the right-click threshold control is
enabled or not.  The SCMTimeValue value controls the delay threshold if
SCMTimeMode is set to True (-1).
```

System Variables

Note: %userprofile% is substituted in the path values below for the sake of clarity and printing only. The actual values use the user profile path as it is expanded on the client machine. Items in green indicate an update since 2002. Items in red indicate a new system variable.

```
ACADVER             "16.0"                              (read only)
```

```
ACISOUTVER          70
CHAMFERA            0.0
CHAMFERB            0.0
CHAMFERC            0.0
CLEANSCREENSTATE    0                                    (read only)
DWGCHECK            0
FILLETRAD           0.0
GFANG               0
GFCLR1              "RGB:000,000,255"
GFCLR2              "RGB:255,255,153"
GFCLRLUM            1.000000000
GFCLRSTATE          1
GFNAME              1
GFSHIFT             0
GRIPHOVER           3
GRIPOBJLIMIT        100
GRIPTIPS            1
INTERSECTIONCOLOR   257
INTERSECTIONDISPLAY OFF
LOCALROOTPREFIX     “%apppath%\...”                   (read only)
MAXSORT             1000
MIRRTEXT            0
MTEXTFIXED          0
MTJIGSTRING         "abc"
MYDOCUMENTSPREFIX   "<%userprofile%>\My Documents"    (read only)
OBSCUREDCOLOR       257                 (range from 0 to 257 only)
OBSCUREDLTYPE       0                   (range from 0 to 11 only)
PALETTEOPAQUE       0
PEDITACCEPT         0
REPORTERROR         1
ROAMABLEROOTPREFIX  "%apppath%\..."                   (read only)
SAVETIME            10
SIGWARN             1
SORTENTS            127
STANDARDSVIOLATION  2
TRAYICONS           1
TRAYNOTIFY          1
TRAYTIMEOUT         0
TSPACEFAC           1.0000
TSPACETYPE          1
VIEWRES             1000
XREFNOTIFY          2
ZOOMFACTOR          60
```

New Commands

3DCONFIG	Configure 3D performance options such as Adaptive Degradation (adjusts resolution with respect to zoom status and viewpoints), as well as many other features used by 3D solids and surfaces drawings
BMPOUT	Export bitmap image file
CLEANSCREENON	Toggle cleanscreen display ON (same as CTRL+0)
CLEANSCREENOFF	Toggle cleanscreen display OFF (same as CTRL+0)
HLSETTINGS	Configure hidden line behavior and display settings
JPGOUT	Export JPEG image file
MREDO	Multiple Redo
PNGOUT	Export Portable Network Graphics (PNG) image file
PUBLISH	Opens Design Publishing utility for publishing DWF files or batch plotting to physical or virtual printer devices
QNEW	Quick-New for creating a new drawing with default options
REVCLOUD	Revision Cloud (formerly part of Express Tools)
SECURITYOPTIONS	Configure digital signature security management options

SETIDROPHANDLER	Configure I-Drop options
SIGVALIDATE	Validate digital signature of current drawing
TIFOUT	Export Tagged Image File format (TIF) image file
TOOLPALETTES	Toggle Tool Palettes ON (same as CTRL+1)
TOOLPALETTESCLOSE	Toggle Tool Palettes OFF (same as CTRL+1)
TRAYSETTINGS	Configure editor system tray settings (display options)
WIPEOUT	WipeOut region creation and editing (formerly part of Express Tools)
XOPEN	XREF open

Modified Commands (Since AutoCAD 2002)

SAVEAS	No more R14 DWG format option
WHOHAS	Enhanced to use the Windows file object handlers and uses a new DWL log file (hidden) to store last-user of file on network.
WMFOUT	Enhanced graphics formatting and display quality
TODAY	Removed!
MEETNOW	Removed!
ENDTODAY	Removed!
VIEWRES	Default is FastRes ON and facet resolution at 1000

Tool Palettes

Tool palettes are an interesting new feature in AutoCAD 2004. Not only do they take the idea of toolbars to a new realm, they introduce XML as a default data storage medium within AutoCAD. Even though DesignXML has been around for quite a while, it was not actually an internally managed data format for storing and retrieving information for routine AutoCAD functionality.

Tool pallets are stored in a folder specified by the *ToolPalettePath* property of the IacadPreferencesFiles collection (see earlier mention). By default, the path is under the user profile path (on Windows 2000 and Windows XP systems, this is under: `\Documents and Settings\<username>\Application Data\Autodesk\AutoCAD 2004\R16.0\<language-code>\Support\ToolPalette\Palettes\` and use the .ATC file extension. The following snippet is taken from the palette file: `Sample office project_00CFCE40-D240-433A-8C62-54E982931ABB.atc`. (You may notice that the beginning part is not a particularly *well-formed* XML document by W3C guidelines, but it gets the job done obviously).

```
<Palette>
  <ItemID idValue="{00CFCE40-D240-433A-8C62-54E982931ABB}"/>
  <Properties>
    <ItemName resource="186" src="AcTpCatalogRes.dll"/><Images/>
    <Time createdUniversalDateTime="2003-02-12T22:37:06" modifiedUniversalDateTime="2003-
02-12T22:37:06"/>
  </Properties>
<Source/>
<Tools>
  <Tool>
    <ItemID idValue="{04314506-7712-4F2C-BB53-B7AD2EA423DD}"/>
    <Properties>
      <ItemName resource="160" src="AcTpCatalogRes.dll"/>
      <Images>
        <Image cx="32" cy="32" resource="IDB_BITMAP_32_OFFICE_CARPET"
src="AcTpCatalogRes.dll"/>
```

```
      </Images>
      <Time createdUniversalDateTime="2003-02-12T22:37:06"
modifiedUniversalDateTime="2003-02-12T22:37:06"/>
    </Properties>
    <Source idValue="{59A63456-0025-4EA0-9CB2-608916D5177C}"/>
    <StockToolRef idValue="{AF0F641B-9CCE-4474-8582-EFE0A38410FC}"/>
    <Data>
      <GeneralProperties>
        <Color>
          <BasicPropInfo>
            <PropValue unspecified="FALSE" valueType="2" value="0"/>
          </BasicPropInfo>
          <CreateInfo>
            <AcCmColor mRGB="3256776847"/>
          </CreateInfo>
        </Color>
        . . . (more) . . .
```

As you can see, this is a typical XML document, however, it does not use the standard XML DTD document header reference you see in most XML formatted documents. The parent document that references all available base palette definition files is the AcTpPalette.atc file, which is located one folder level above the folder used by palette definition files stated above.

If you Export a palette, it is written in XML form to a .XTP file. The default folder location is the My Documents folder path. Unlike some other AutoCAD export functions, this one actually uses the name of the palette as the default base filename value. For example, exporting the "Sample Office Project" will default to "sample office project.xtp" in the My Documents folder, unless you specify otherwise.

An interesting thing to note here is that a new command has been added to act as a wrapper to the previous TOOLBAR command. The new command for creating, modifying, importing and exporting toolbar button files and palettes is CUSTOMIZE. You can invoke it by right-clicking on a palette and picking the Customize option, or by typing it in at the command prompt. Below is the "Sample Office Project" palette export file (snippet only) to show what an XTP file looks like internally. Pay particular attention to how the insertion properties are represented <data> section colored green below.

```
<ToolPaletteExport>
  <DependentFilesPath>Sample office Project</DependentFilesPath>
  <PaletteSets>
    <PaletteSetData>
      <PaletteSetContent>
        <Palettes>
          <PaletteData>
            <PaletteContent>
              <Palette>
                <ItemID idValue="{00CFCE40-D240-433A-8C62-54E982931ABB}"/>
                <Properties>
                  <ItemName resource="186" src="AcTpCatalogRes.dll"/>
                  <Images/>
                </Properties>
                <Source/>
                <Tools>
                  <Tool>
                    <ItemID idValue="{04314506-7712-4F2C-BB53-B7AD2EA423DD}"/>
                    <Properties>
                      <ItemName resource="160" src="AcTpCatalogRes.dll"/>
                      <Images>
```

```
                        <Image cx="32" cy="32" resource="IDB_BITMAP_32_OFFICE_CARPET"
src="AcTpCatalogRes.dll"/>
                      </Images>
                      <Time createdUniversalDateTime="2003-02-12T22:37:06"
modifiedUniversalDateTime="2003-02-12T22:37:06"/>
                    </Properties>
                    <Source idValue="{59A63456-0025-4EA0-9CB2-608916D5177C}"/>
                    <StockToolRef idValue="{AF0F641B-9CCE-4474-8582-EFE0A38410FC}"/>
                    <Data>
                      <GeneralProperties>
                        <Color>
                          <BasicPropInfo>
                            <PropValue unspecified="FALSE" valueType="2" value="0"/>
                          </BasicPropInfo>
                          <CreateInfo>
                            <AcCmColor mRGB="3256776847"/>
                          </CreateInfo>
                        </Color>
                        <Layer>
                          <BasicPropInfo>
                            <PropValue unspecified="TRUE" valueType="1"/>
                          </BasicPropInfo>
                        </Layer>
                        <Linetype>
                          <BasicPropInfo>
                            <PropValue unspecified="TRUE" valueType="1"/>
                          </BasicPropInfo>
                        </Linetype>
                        <LinetypeScale>
                          <BasicPropInfo>
                            <PropValue unspecified="TRUE" valueType="3" value="0"/>
                          </BasicPropInfo>
                        </LinetypeScale>
                        <PlotStyle></PlotStyle>
                        <LineWeight></LineWeight>
                        <HatchType></HatchType>
                        <PatternName></PatternName>
. . .
```

Conclusion

While this book covers a lot of information about using Visual LISP, it cannot cover everything. Visual LISP has a lot of potential and provides a lot of powerful tools to LISP developers. I wish Autodesk would commit some resources to improving it to bring it up to speed with current development tools. At the very least, fix some of the incomplete features and irritating quirks, but I think Autodesk is letting LISP die on the vine in favor of VBA and ARX. That would be unfortunate and a big mistake as well.

Most of the features that could be improved would require very minimal investment in time and budget and would yield a much more robust development tool. Some improvements might be fixing the project management tools, better compilation controls, fixing dialog box inconsistencies, adding "Intelli-Sense" completion, streamlined functionality for referencing external objects and components, making a standalone version and so on.

Hopefully, the information and examples provided herein will give you some additional motivation to further explore Visual LISP and become a better software developer as a result. If not, it makes a great coffee cup stand. In any case, I hope you find this book useful and helpful.

Happy Coding!

Dave

Appendix A - VLAX Enumeration Constants

Constant Symbol	**Value**
:vlax-false	:vlax-false
:vlax-null	:vlax-null
:vlax-true	:vlax-true
vlax-vbAbort	3
vlax-vbAbortRetryIgnore	2
vlax-vbApplicationModal	0
vlax-vbArchive	32
vlax-vbArray	8192
vlax-vbBoolean	11
vlax-vbCancel	2
vlax-vbCritical	16
vlax-vbCurrency	6
vlax-vbDataObject	13
vlax-vbDate	7
vlax-vbDefaultButton1	0
vlax-vbDefaultButton2	256
vlax-vbDefaultButton3	512
vlax-vbDirectory	16
vlax-vbDouble	5
vlax-vbEmpty	0
vlax-vbError	10
vlax-vbExclamation	48
vlax-vbHidden	2
vlax-vbHiragana	32
vlax-vbIgnore	5
vlax-vbInformation	64
vlax-vbInteger	2
vlax-vbKatakana	16
vlax-vbLong	3
vlax-vbLowerCase	2
vlax-vbNarrow	8
vlax-vbNo	7
vlax-vbNormal	0
vlax-vbNull	1
vlax-vbObject	9
vlax-vbOK	1
vlax-vbOKCancel	1
vlax-vbOKOnly	0
vlax-vbProperCase	3
vlax-vbQuestion	32
vlax-vbReadOnly	1
vlax-vbRetry	4
vlax-vbRetryCancel	5
vlax-vbSingle	4
vlax-vbString	8
vlax-vbSystem	4
vlax-vbSystemModal	4096
vlax-vbUpperCase	1
vlax-vbVariant	12
vlax-vbVolume	8
vlax-vbWide	4
vlax-vbYes	6

vlax-vbYesNo	4
vlax-vbYesNoCancel	3

Appendix B – VLISP IDE Keyboard Shortcuts

F1	Help
F3	Find / Replace Next
F6	Display LISP Console Window
F8	Step Into
SHIFT+F8	Step Over
CTRL+SHIFT+F8	Step Out Of
F9	Toggle BreakPoint
CTRL+SHIFT+F9	Clear All BreakPoints
CTRL+W	Add Watch
CTRL+R	Reset to Current Level
CTRL+Q	Quit to Current Level
ALT+F6	Zoom
ALT+Q	Exit / Quit Visual LISP IDE

Appendix C – Tips & Tricks for Visual LISP

Adding VLX support to the (autoload) function

By default, the AutoLISP (autoload) function will only look for .LSP, .FAS or .MNL file types when searching for a specified file loading. It will not consider .VLX files at all. This is a minor but often significant oversight by Autodesk, but fortunately it is an easy fix. Simply open the Acad2000doc.lsp file located in the Support folder of your AutoCAD installation. Then locate the (defun ai_ffile) function definition and add an additional check for .vlx files, save the file and close it. This will need to be done on every machine that you wish to perform this change.

Saving your VLIDE configuration settings

When you modify the editor configuration settings your changes are saved in a configuration file named VLIDE.DSK in the Support folder where your local copy of AutoCAD (or network client) is installed. You should keep a copy of this file elsewhere to avoid it being overwritten when reinstalling AutoCAD or installing a service pack update. This file contains your formatting preferences (colors, tabs, indentation, etc.).

Recovering DCL Code from VLX Files

In Chapter 13 the details of how LSP and DCL are compiled into VLX output was discussed. One thing to remember is that while the LSP code is first compiled into FAS form before being compiled into the VLX output, DCL code is not compiled at all. It is simply appended to the bottom of the VLX output file. Therefore, you can open VLX files in any standard text editor such as Windows Notepad, and browse to the bottom of the file to find all the DCL code in tact. You can copy and paste it from there back into a new DCL file whenever you delete a DCL file by mistake but happen to have the VLX available.

Using Projects and DCL with the Make Application Wizard

To put Projects to their full use, it is a good practice to add all related files for a given VLX into a project. Then order them in the proper sequence based upon the order of function definition statements (load order). Next, when using the Make Application wizard, select the Project PRJ file instead of the individual LSP files.

Since DCL files cannot be included in a Project list, you would normally have to add them individually in the Resource files list of the Make Application wizard (Expert mode only). However, another approach can be to concatenate all the DCL files into a single DCL file. This will result in having to add one PRJ and one DCL file to make the VLX application.

To concatenate multiple DCL files into a single DCL file, use the age-old DOS command COPY as follows:

```
    COPY *.DCL ALL.DCL
```

This will copy all .DCL files into a single file named ALL.DCL. Remember to consider the relative paths, or run everything from within the same path location.

Team-based VLX Development

If you're developing a network based project along with several other developers, you should keep a few things in mind. First, the PRV format stores the path/drive information for your source files when compiling VLX applications. Secondly, sharing PRV setup files only works when the relative pathing is portable to all intended users. By "portable", I mean that the relative path/drive information must be equally applicable by all intended users. It's a good idea to have all the developers copy the entire source code set to their local hard drives and use local pathing for all your PRV file setups. Then after the VLX apps are built, copy them up to the intended network server(s).

In addition, if you have some sort of change management software, such as Visual Source Safe, StarBase, PVCS-DOORS or whatever, you should use that in order to manage check-out, check-in and version control over all the various files to avoid stepping on each other's work and creating confusion.

Appendix D – Useful Resources

Helpful/Recommended Web Sites:

http://www.acadx.com **

http://www.vbdesign.net **

http://www.upfront.com

http://www.ntfaq.com

http://www.microsoft.com/scripting

http://www.microsoft.com/data

http://www.tenlinks.com

http://www.cadinfo.net

http://www.swynk.com

http://www.adminscripts.net

http://www.fourguysfromrolla.com

http://www.15seconds.com

http://www.win2000mag.net

http://www.planet-source-code.com

Glossary

ActiveX	(boy, oh boy. Ask Microsoft)
Bookmark	A location marker placed in a document that enables the user to return to that location quickly.
Breakpoint	A marker placed in program code that instructs the compiler or interpreter to pause execution at runtime and wait for the user to perform debugging tasks or continue execution.
Call-Back	A requested response to a given action or event. For example, a call-back to clicking a button might be "accept" which then triggers a call to a particular function or expression.
Collection	A group of objects with a common parent and related properties or methods that enable processing the objects in a logical manner as a group.
COM	Component Object Model. A Microsoft technology that defines a hierarchical organization of software components, and services and provides intrinsic properties, methods and events for components and services that enable more efficient programmatic manipulation and promotes componentized functional reuse. Other flavors include Distributed COM or DCOM and the newer COM+ included with Windows 2000 and XP platforms.
Constant	A variable or symbol with a static value assignment.
Consumer	Any software component or application that imports or uses the exposed component services of another software component or service. The source of the imported components or services is known as a *provider*.
Control	An ActiveX DLL component.
Data Type	The intrinsic nature of a particular value with respect to what form of data is represented. Examples of ActiveX data types include Integer, Long, Double, String, and Array.
DCL	Dialog Control Language, a C-based language construct used to define dialog box forms within the AutoCAD LISP and Visual LISP environments.
Debug	The process of isolating, diagnosing and correcting errors in program code or programming logic.

Dictionary	A type of collection that provides direct access to member objects by a using unique identifier for each object.
DLL	A *dynamic link library* is a Windows-based ActiveX component that usually exposes functions, properties, methods, and constants for use by other applications. It is something like a packaged library of tools that can be loaded by applications to perform specialized tasks.
Element	An individual member of an array or safearray construct.
Enumeration	(need an official definition for this one)
Evaluate	The process of executing a LISP expression or extracting an associated value from a LISP symbol and returning a result.
Event	A moment when some action occurs in a software program. This can be the click of a button or moving an entity. An event usually provides programmatic notification that can be detected and responded to using a *reactor* or *callback*.
Expression	A program statement within the context of AutoLISP or Visual LISP interpreter environments.
Focus	The state of a given item within a DCL dialog box either having control by the active cursor location. If an editbox has the cursor active and is editable, it is said to *have the focus*. When the cursor is moved out of a given item, it is said to have *lost the focus*.
Function	In the context of software development, this is an expression or group of expressions that processes some type of input and returns a result. In the context of Visual LISP, a *subroutine* and a *function* are synonymous. In the context of other languages like Visual Basic or C/C++ a *subroutine* returns no result, while a *function* returns a result.
Global	Any variable, symbol or expression that is exposed for either read or write manipulation by all other variables, symbols or expression running in the same namespace. A symbol or expression that is hidden from access by other expressions is said to be Localized or Local to its parent function or expression.
Heap	A pile of trash that needs to be cleaned up, or a logical memory address space allocated for a particular group of related expression definitions and/or their results.
Interface	(Programming) any means by which one software component or service can connect to another for the purposes of requesting

	services or values from the other component or service. This can also involve the passing of information in either direction through a common logical programmatic reference.
Iterate	A loop process where items are accessed in a sequential order within a group of related items. Examples of iteration functions are (while), (do while), (foreach), (repeat), and (vlax-for).
Local	Any variable, symbol or expression that is not shared or accessible outside of its parent function or expression.
Marshalling	The process of controlling an external process remotely from another process. Launching another application in serialized fashion is but one example of marshalling.
Method	A built-in function of a given object that enables automated retrieval or modification of that object. Examples of methods provided by a Line object include Move, Rotate, and Copy.
Modal / Modeless	Refers to the nature of how a dialog form can be displayed and controlled within the environment in which it is launched. If the dialog can remain visible while the user can continue to interact with other aspects of the parent application, the form is said to be Modeless. If the visible form prevents interaction with other aspects of the parent application until it is closed, that form is said to be Modal in nature.
NameSpace	An isolated memory address range allocated to a given application or process. The address range is protected from access by other address ranges, and thereby creates a protected environment for the process to execute within.
Object	In the context of software development: An instance of a class that provides intrinsic functionality such as properties, methods or events that can be used to interact with other services, components or objects to perform some programming task.
Object Model	A logical, hierarchical organization of objects within a parent software application or process. For example, Windows 2000 has an object model provided by the Win32 and DCOM or COM+ class environments. AutoCAD 2002 has its own object model provided by the ActiveX framework and exposed through ObjectDBX, ObjectARX, VBA and Visual LISP environments.
Project	(Visual LISP) a named collection of program source code files.

Property	An intrinsic attribute of a given object that enables unique identification of that object in some way.
Provider	A software application or component that exposes some functionality for use by other applications or components. In particular, it becomes a provider while it is actually used by a *consumer* application, component or service.
Reactor	A special type of software service provided by Visual LISP that acts as a listening device for specific events within the AutoCAD application session and optionally performs some task when a specified condition is met by an intercepted event.
Recursion	The process of evaluating inputs to a given function whereby the repetitive processing to achieve a desired result involves one or more self-invocations of the same function by itself until a final, terminating condition is met.
RPC	A Remote Procedure Call process that is provided by the Windows operating system to allow a local process to request a process to be created on another machine to perform some action remotely.
SafeArray	An array of elements whereby the array has a fixed length and cannot be modified to increase or decrease the length (number of elements that can be stored within it). It is said to be “safe” because it cannot change length, and thereby reduces the possibility of errors as a result of attempting to enter or retrieve elements from an index that is *out of bounds* (beyond the end of the array).
Scope	A minty mouthwash, or a logical boundary with respect to the reach or lifespan of a specific symbol or expression
Stack	A logical container that can collect objects or values and allow for systematic addition or removal of members in an orderly or sequential manner.
Stepping	The process of stopping program execution and allowing the user to manually advance execution one statement or one line at a time. There are several types of stepping: Step Into, Step Over and Step Out Of.
Type Casting	The process of converting one data type to another, such as converting an integer number value to a string value.
Type Library	Also called a TypeLib, is a dedicated software component that identifies the object model and member objects, properties,

	methods, and constants of that object model to any other applications or processes that request programmatic interaction.
Variant	A data type that is defined to be capable of storing all other data types, thereby avoiding the concern of verifying a given data type before assigning or retrieving it from a given variable or symbol.
WorkSpace	The active set of related program documents opened in the programming development environment.
WSH	Microsoft’s Windows Scripting Host, a software service that provides runtime support for executing script code files on a machine. WSH allows scripts to be run either in the Windows namespace or within the namespace of a calling application or process from a programmatic interface. The default WSH enabling services are CSCRIPT (command line interface) and WSCRIPT (graphical user interface).