AfraLisp

The

AutoLisp

Tutorials

Visual Lisp

Written and Compiled by Kenny Ramage
afralisp@mweb.com.na
http://www.afralisp.com

Contents

Application (Global)

- Preferences
- Documents
  - Document
    - Blocks → Block → Database
    - ModelSpace
    - Paperspace → PViewport
    - Dictionaries → Dictionary
      - XRecord
    - DimStyles → DimStyle
    - Groups → Group
    - Layers → Layer
    - Layouts → Layout
    - Linetypes → Linetype
    - PlotConfigurations → PlotConfiguration
    - RegisteredApps → RegisteredApplication
    - SelectionSets → SelectionSet
    - TextStyles → TextStyle
    - UCSs → UCS
    - Views → View
    - Viewports → Viewport

Entities (blue)

- 3DFace
- 3DPoly
- 3DSolid
- Arc
- Attribute
- AttributeRef
- BlockRef
- Circle
- Dim3PointAngular
- DimAligned
- DimAngular
- DimDiametric
- DimOrdinate
- DimRadial
- DimRotated
- Ellipse
- ExternalReference
- Hatch
- Leader
- LightweightPolyline
- Line
- MInsertBlock
- MLine

Preferences

- PreferencesDisplay
- PreferencesDrafting
- PreferencesFiles
- PreferencesOpenSave
- PreferencesOutput
- PreferencesProfiles
- PreferencesSelection
- PreferencesSystem
- PreferencesUser

Hyperlinks

- Hyperlink

Viewports

Viewport

DatabasePreferences

Plot

Utility

MenuBar

PopupMenu

MenuGroups

MenuGroup

PopupMenus

PopupMenu

PopupMenuItem

Toolbars

Toolbar

ToolbarItem

MLine

MText

Point

PolyfaceMesh

Polyline

PolygonMesh

Raster

Ray

Region

Shape

Solid

Spline

Text

Tolerance

Trace

XLine

LEGEND

Color

Database resident entity
(supports IAcadEntity and IAcadObject)
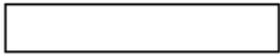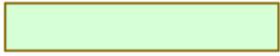
Database resident object
(supports IAcadObject)

Non-database resident

Shape

Collection

Object

Using the Visual Lisp Editor

This tutorial is a crash course in using the Visual Lisp Editor, and is not intended to be detailed or fully comprehensive. The aim is to show you the main functions of the Editor with the intention of getting you up and running as quickly as possible.
Right, enough waffle, let's get started. Fire up AutoCAD and open a new drawing.
Now choose "TOOLS" - "AUTOLISP" - "VISUAL LISP EDITOR".
The Visual Lisp Editor will open and should look like this :
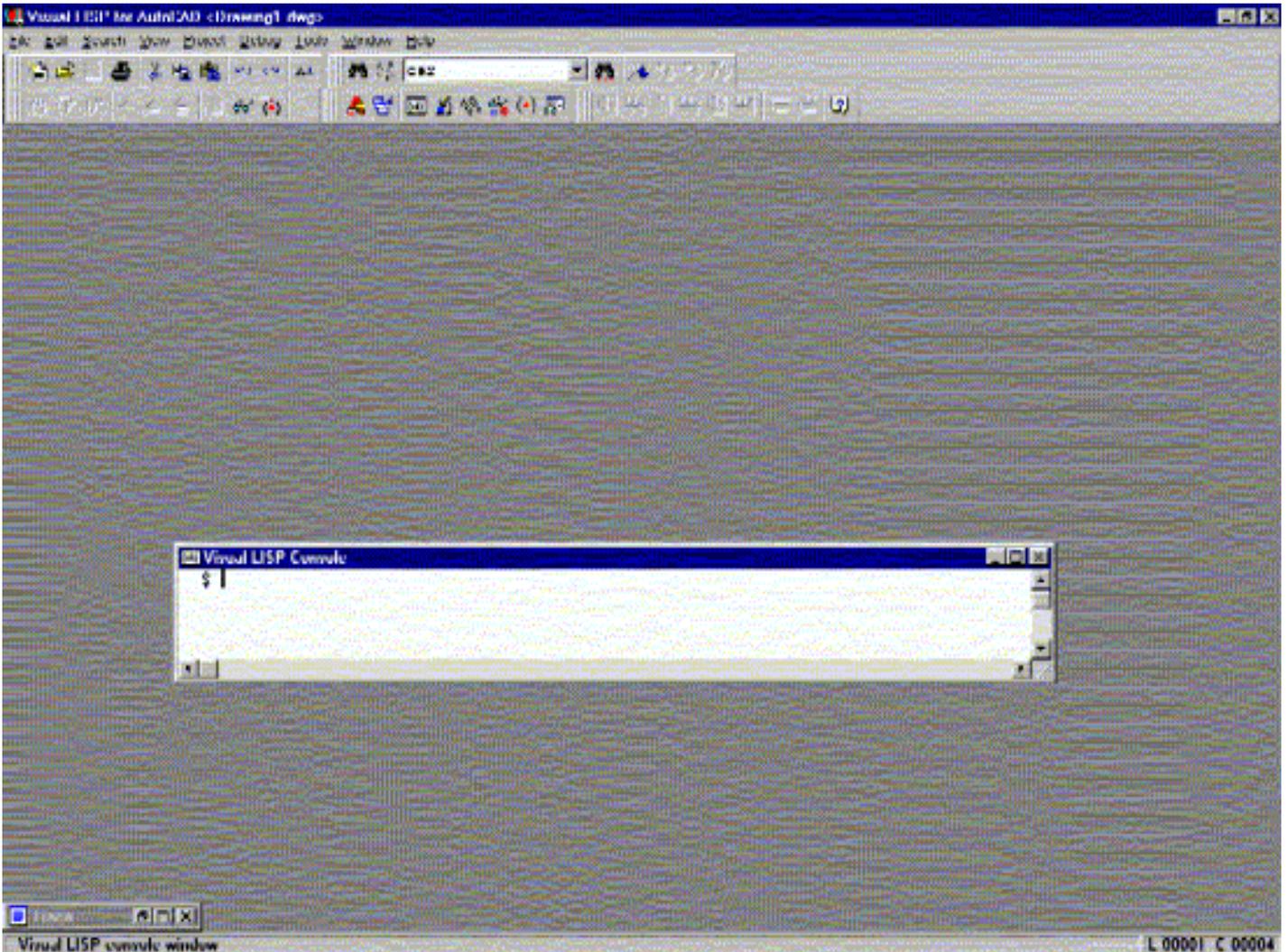
Let's start off by having a look at the Console Window :
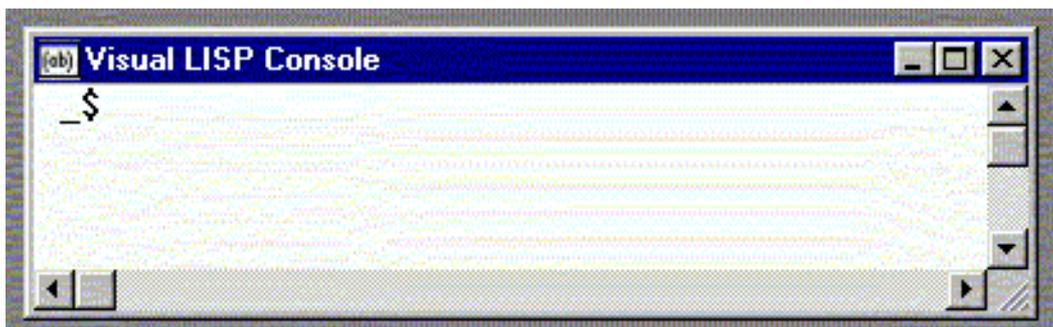
The VLISP Console window is similar in some respects to the AutoCAD Command window, but has a few extra features. You enter text into the Console window following

the Console prompt which looks like this :

_$

Type this in the Console prompt and then press "Enter" :

_$ (setq a "Test")

Now type this and again press "Enter" :

_$ a

Your Console window should look like this :
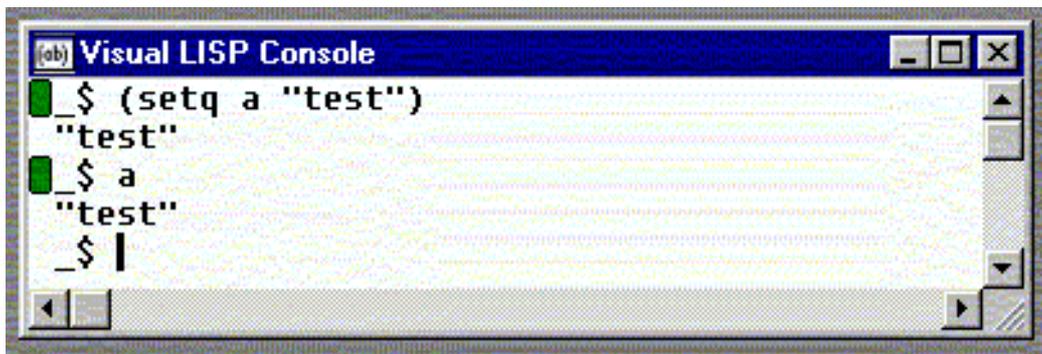
To view the value of a variable at the AutoCAD Command prompt, you must precede the variable name with an exclamation mark. (!) In VLISP, you simply type the variable name.

Unlike the AutoCAD Command window, where pressing SPACEBAR causes expression evaluation, text input at the VLISP Console prompt is not processed until you press ENTER. This permits you to do the following in the Console window:

- Continue an AutoLISP expression on a new line. To continue entering an expression on a new line, press CTRL +ENTER at the point you want to continue.

- Input more than one expression before pressing ENTER. VLISP evaluates each expression before returning a value to the Console window.

- If you select text in the Console window (for example, the result of a previous command or a previously entered expression), then press ENTER, VLISP copies the selected text at the Console prompt.

The VLISP Console window and the AutoCAD Command window differ in the way they process the SPACEBAR and TAB keys. In the VLISP Console window, a space plays no special role and serves only as a separator. In the AutoCAD Command window, pressing the SPACEBAR outside an expression causes AutoCAD to process the text immediately, as if you had pressed ENTER.

Using the Console Window History

You can retrieve text you previously entered in the Console window by pressing TAB while at the Console prompt. Each time you press TAB, the previously entered text replaces the text at the Console prompt. You can repeatedly press TAB until you cycle through all the text entered at the Console prompt during your VLISP session. After you've scrolled to the first entered line, VLISP starts again by retrieving the last command entered in the Console window, and the cycle repeats. Press SHIFT + TAB to scroll the input history in the opposite direction. For example, assume you entered the following commands at the Console prompt:

(setq origin (getpoint "\nOrigin of inyn sign: "))

(setq radius (getdist "\nRadius of inyn sign: " origin))

(setq half-r (/ radius 2))

(setq origin-x (car origin))

(command "_.CIRCLE" origin radius)

To retrieve commands entered in the Console window

1 Press TAB once. VLISP retrieves the last command entered and places it at the Console prompt:

$ (command ".CIRCLE" origin radius)

2 Press TAB again. The following command displays at the Console prompt:

_$ (setq origin-x (car origin))

3 Press TAB again. VLISP displays the following command:

_$ (setq half-r (/ radius 2))

4 Now press SHIFT+ TAB . VLISP reverses direction and retrieves the command you entered after the previous command:

_$ (setq origin-x (car origin))

5 Press SHIFT+ TAB again. VLISP displays the following command:

$ (command ".CIRCLE" origin radius)

This was the last command you entered at the Console prompt.

6 Press SHIFT+ TAB again. Because the previous command retrieved was the last command you entered during this VLISP session, VLISP starts again by retrieving the first command you entered in the Console window:

_$ (setq origin (getpoint "\nOrigin of inyn sign: "))

Note that if you enter the same expression more than once, it appears only once as you cycle through the Console window input history. You can perform an associative search in the input history to retrieve a specific command that you previously entered. To perform an associative search of the Console input history

1 Enter the text you want to locate. For example, enter *(command* at the Console prompt:

_$ (command

2 Press TAB. VLISP searches for the last text you entered that began with *(command*:

$ (command ".CIRCLE" origin radius)

If VLISP does not find a match, it does nothing (except possibly emit a beep). Press SHIFT+ TAB to reverse the direction of the associative search and find progressively less-recent inputs.

Interrupting Commands and Clearing the Console Input Area

To interrupt a command entered in the Console window, press SHIFT + ESC. For example, if you enter an invalid function call like the following:

_$ ((setq origin-x (car origin)

((_>

Pressing SHIFT + ESC interrupts the command, and VLISP displays an "input discarded" message like the following:

((_> ; <input discarded>

_$

If you type text at the Console prompt, but do not press ENTER, then pressing ESC clears the text you typed. If you press SHIFT + ESC, VLISP leaves the text you entered in the Console window but displays a new prompt without evaluating the text.

If you type part of a command at the Console prompt, but activate the AutoCAD window before pressing ENTER, VLISP displays a new prompt when you next activate the VLISP window. The text you typed is visible in the Console window history, so you can copy

and paste it, but you cannot retrieve the text by pressing TAB , because it was not added to the Console history buffer.

Right, enough messing about. Let's load some coding. Choose "File" - "New" and then copy and paste this coding into the text editor :

```
(defun C:SLOT ( )

  (setvar "CMDECHO" 0)
  (setvar "BLIPMODE" 0)
  (setq oldsnap (getvar "OSMODE"))

  (setq diam (getdist "\nSlot Diameter : ")
        lngth (getdist "\nSlot Length : "))

  (while
    (setq pt1 (getpoint "\nInsertion point: "))
    (setvar "OSMODE" 0)
    (setq pt2 (polar pt1 0.0 (/ (- lngth diam) 2.0))
          pt3 (polar pt2 (/ pi 2.0) (/ diam 4.0))
          pt4 (polar pt3 pi (- lngth diam))
          pt5 (polar pt4 (* pi 1.5) (/ diam 2.0))
          pt6 (polar pt5 0.0 (- lngth diam)))

    (command "PLINE" pt3 "W" (/ diam 2.0) "" pt4
             "ARC" pt5 "LINE" pt6 "ARC" "CLOSE")
    (setvar "OSMODE" oldsnap)
  );while
  (princ)
);defun
(princ)
```

The coding should look like this in the text editor :

```
slot.lsp                                                    _ □ ✕
(defun C:SLOT ()

    (setvar "CMDECHO" 0)
    (setvar "BLIPMODE" 0)
    (setq oldsnap (getvar "OSMODE"))

    (setq diam (getdist "\nSlot Diameter : ")
          lngth (getdist "\nSlot Length : "))

    (while
        (setq pt1 (getpoint "\nInsertion point: "))
        (setvar "OSMODE" 0)
        (setq pt2 (polar pt1 0.0 (/ (- lngth diam) 2.0))
              pt3 (polar pt2 (/ pi 2.0) (/ diam 4.0))
              pt4 (polar pt3 pi (- lngth diam))
              pt5 (polar pt4 (* pi 1.5) (/ diam 2.0))
              pt6 (polar pt5 0.0 (- lngth diam)))

        (command "PLINE" pt3 "W" (/ diam 2.0) "" pt4
                 "ARC" pt5 "LINE" pt6 "ARC" "CLOSE")
        (setvar "OSMODE" oldsnap)
    );while
    (princ)
);defun
(princ)
```

Before we go any further, let's have a wee chat about the colors.
As soon as you enter text in the VLISP Console or text editor windows, VLISP attempts to determine if the entered word is a built-in AutoLISP function, a number, a string, or some other language element. VLISP assigns every type of element its own color. This helps you detect missing quotes or misspelled function names. The default color scheme is shown in the following table.

AutoLISP Language Element	Color
Built-in functions and protected symbols	Blue
Strings	Magenta
Integers	Green
Real numbers	Teal
Comments	Magenta, on gray background
Parentheses	Red
Unrecognized items (for example, user variables)	Black

You can change the default colors. But, do yourself a favour. Don't!!

Selecting Text

The simplest method to select text is to double-click your left mouse button. The amount of text selected depends on the location of your cursor.

- If the cursor immediately precedes an open parenthesis, VLISP selects all the following text up to the matching close parenthesis.

- If the cursor immediately follows a close parenthesis, VLISP selects all preceding text up to the matching open parenthesis.

- If the cursor immediately precedes or follows a word, or is within a word, VLISP selects that word.

Hint : Would you like help on any AutoLisp function? Double-click on the function name to select it, and then select the "Help" toolbar button. Help for the specific function will be displayed.

To load the "Slot" lisp routine, select the "Load active edit window" toolbar button :
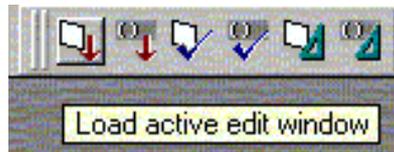
This loads your coding into memory. To run the routine, type this at the Console prompt :

_$ (c:slot)

The program should now run, switching over to the AutoCAD screen when required. You can also just run a selection of code if you desire. Select the lines of code you would like to run and choose the "Load selection" toolbar button, and then press "Enter."

Only the lines of code you selected will be run, Great for debugging.

Talking about debugging, Place the cursor in front of the *(defun C:SLOT ()* statement and press "F9". This places a "Breakpoint" into your program. Now run the program again. Execution should stop at the Breakpoint mark. Now press "F8". By continuously pressing "F8" you can "single step" through your whole program :
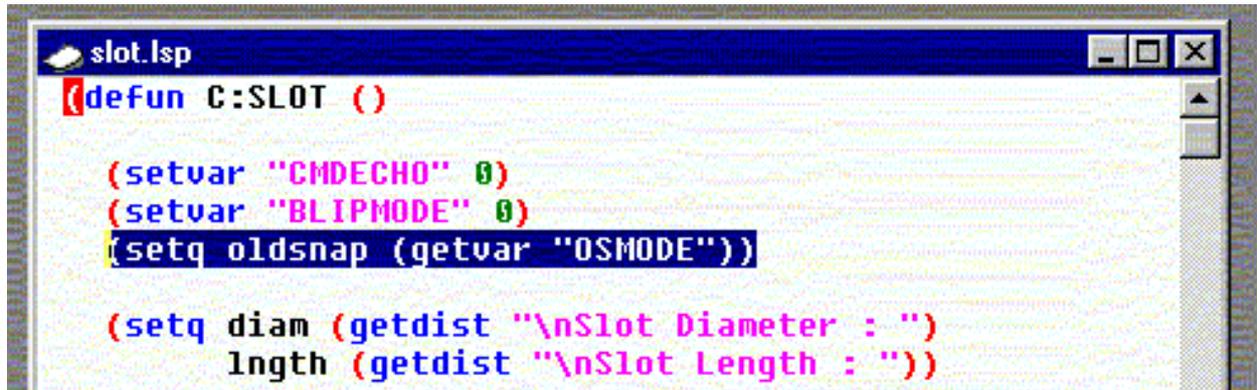
Let's go one step further. Select the "diam" variable and then select the "Add Watch" toolbar button :

The "Watch" dialog box will appear :
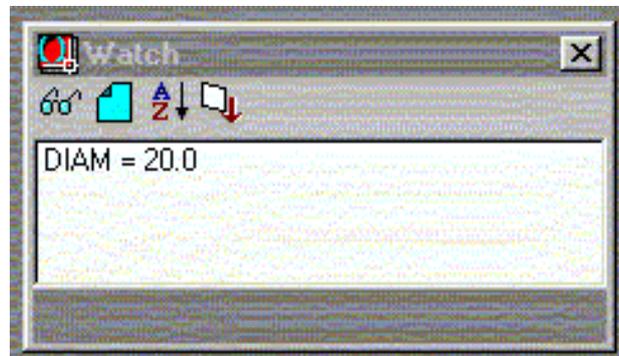
Note how the variable "diam" is listed along with it's present value. Repeat this process for all the other variables until the Watch" dialog looks like this :
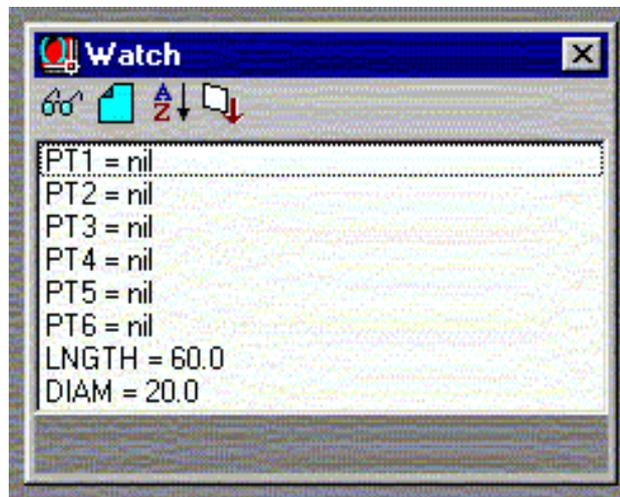
Now run the program again, still "single" stepping through. Notice how the values of the variables change as the program proceeds.

O.K. let's liven things up a bit. Select "Ctrl-Shift-F9" to clear all breakpoints. Now select the "Debug" pull down menu and then "Animate". Now run the program again.
Hey, it's running automatically!!! Take note of the variables changing in the "Watch" window as the program does it's thing.

Well that's about it in regards to the Visual Lisp Editor. The editor has a lot more functions than I've shown you here, but these I feel, are some of the more important ones to get you started.

The Beginning.

So, you want to start coding using Visual Lisp? Two things. First you really need to have a good understanding of AutoLisp before you carry on with this Tutorial. VLisp is not an replacement for AutoLisp, it is an extension to it. Standard AutoLisp is used extensively throughout Visual Lisp, so a good knowledge is a necessity. Secondly, in this tutorial I am not going to delve deep into the why's and where's of VLisp. The intention is to give you a basic grounding into what VLisp can do and how to go about doing it. For example, some of my terminology may not be technically correct as I've tended to convert some things to layman terms for clarity and ease of understanding. Don't worry, we'll correct all that in future tutorials.

O.K. are you ready to start? Right, fire up AutoCAD with a blank drawing and open the Visual Lisp Editor. You can write Visual Lisp using Notepad just like AutoLisp if you wish, but I prefer to use the Visual Lisp Editor as we can, and will, use the "Watch" and the "Inspect" windows.
Close the Editor window, leaving the "Console" window open, and then open the "Watch" window. Right, we're ready to start.
Type this at the Console prompt and then press enter :

_$ (vl-load-com)

Did you notice that nothing happened? Before you can use the VLisp functions with AutoLisp, you need to load the supporting code that enables these functions. The (vl-load-com) function first checks if the VLisp support is already loaded; if so, the function does nothing. If the functions are not loaded, (vl-load-com) loads them. Pretty important I would say!

"All applications that use Visual Lisp should begin by calling (vl-load-com). If (vl-load-com) is not loaded, the application will fail."

After loading the Visual Lisp functions, the next step is to establish communication with the AutoCAD Application object.
Think of a filing cabinet. The cabinet itself is your computer and one of the filing cabinet drawers is AutoCAD. We are going to open the AutoCAD drawer to have a look at what is inside.
To establish this connection, we use the (vlax-get-acad-object) function.
Type this at the Console prompt and then press enter :

_$ (setq acadObject (vlax-get-acad-object))

Now, double click on the variable "acadObject" in the Console window to select it, and then add it to the "Watch" window. It should look something like this :
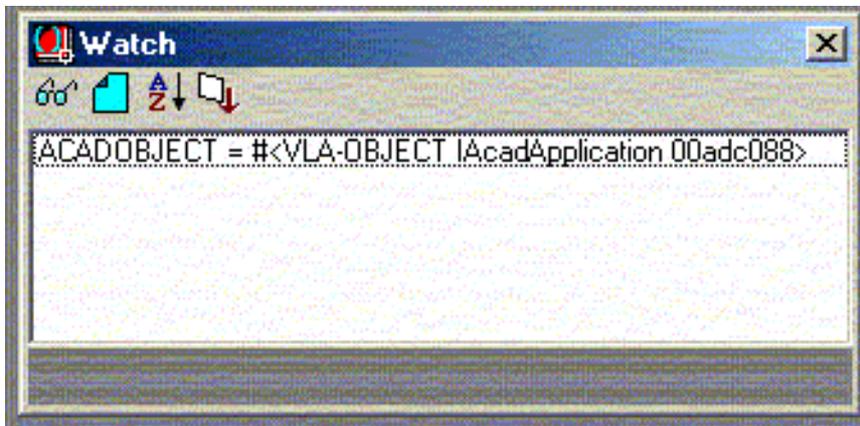
We are now looking at the Acad Application object. Double click on the "ACODBJECT" in the "Watch window to open the "Inspect" window :
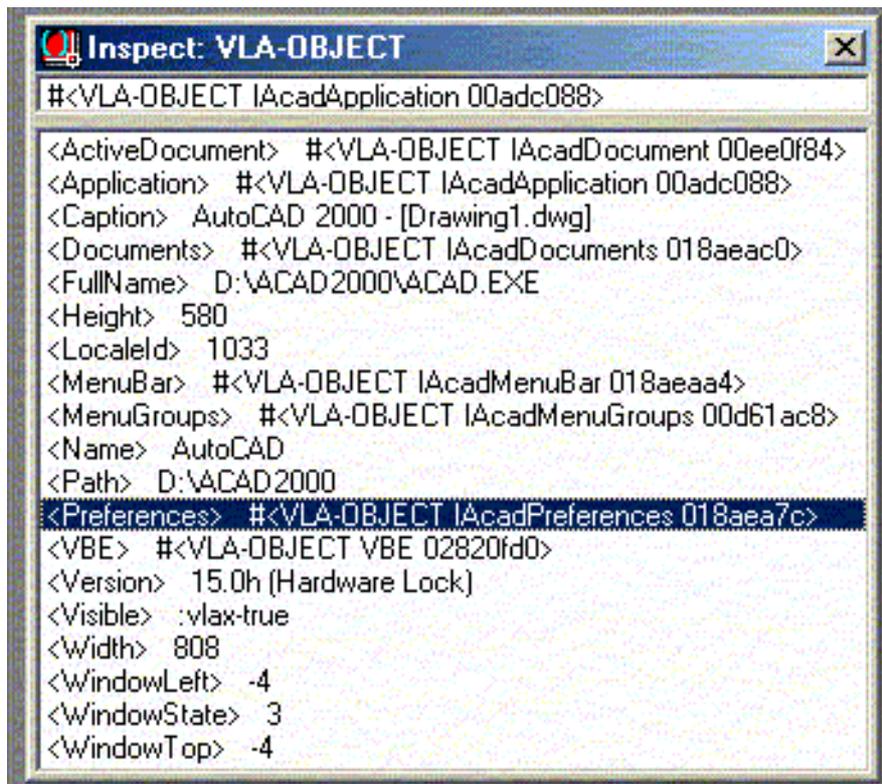
This is a list of all the Objects within the AutoCAD Application Object.

Now double click on <Preferences> :
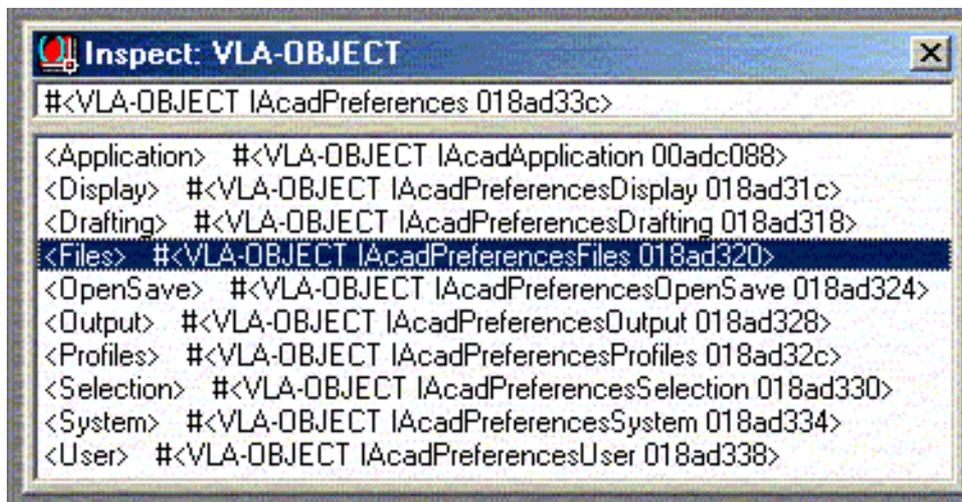
This is all the Objects within the AutoCAD - Application - Preferences object . Now double click on <Files> :

Compare this list with what you see when you open the Options-Files Dialog :
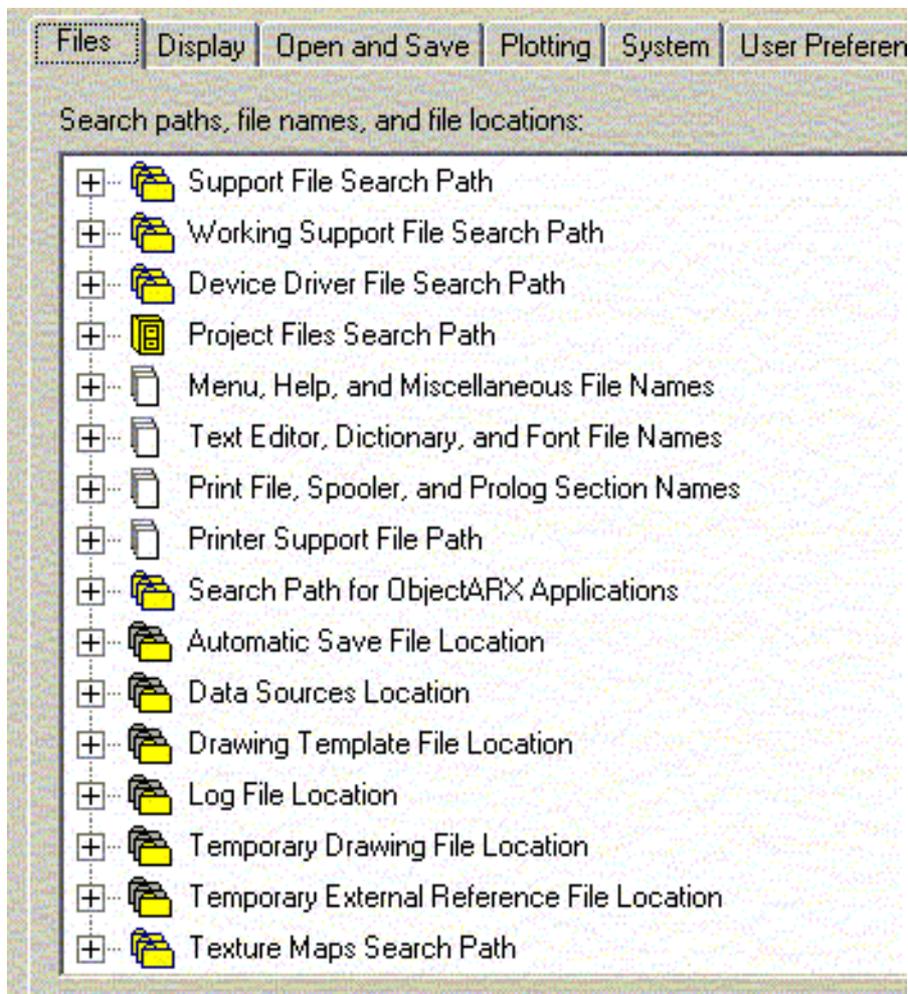
Pretty much the same hey? On the next page we'll have a look at how we can access these Objects programatically.

Before we start this section, I think it might be a good idea if you pop along and have a look at the AutoCAD Object Model - You will find this on Page 3. In fact, print it out and have it next to you as you work through these tutorials. It's just like having a road map.

Anyway, where were we? Oh, yes. We are now going to try and extract the AutoCAD Support Path from the Object Model using VLisp. We'll take it right from the top just in case you got a bit lost. Here we go. Type all of the following statements into the Console window, pressing "Enter" after each one :
(The lines in red are what VLisp returns, don't type them!)

Load the VLisp support
_$ (vl-load-com)

Store a reference to the Application Object
_$ (setq acadObject (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00adc088>

Store a reference to the Preferences Object
_$ (setq prefsObject (vlax-get-property acadObject 'Preferences))
#<VLA-OBJECT IAcadPreferences 018adfdc>

Store a reference to the Files Object
_$ (setq tabnameObject (vlax-get-property prefsObject 'Files))
#<VLA-OBJECT IAcadPreferencesFiles 018adfc0>

Get the Support Path
_$ (setq thePath (vlax-get-property tabnameObject 'SupportPath))
"D:\\ACAD2000\\SUPPORT;D:\\ACAD2000\\FONTS;
D:\\ACAD2000\\HELP;D:\\ACAD2000\\EXPRESS"

"Jings, that's great, but can we add a new directory to our Support Path?"
Of course we can, but first we need to add our new path to the existing path variable. What should we use? Easy, let's just use the *(strcat)* function :

_$ (setq thePath (strcat thePath ";" "C:\\TEMP"))
"D:\\ACAD2000\\SUPPORT;D:\\ACAD2000\\FONTS;
D:\\ACAD2000\\HELP;D:\\ACAD2000\\EXPRESS;C:\\TEMP"

We've added the new directory to the Support Path variable, so now let's update it. Type in this line:

_$ (vlax-put-property tabnameObject 'SupportPath thePath)
nil

Now, return to AutoCAD and go to *"Tools" - "Options" - "Files" - "Support Search Path"*. Your new directory, C:/TEMP, should have been added to your Support Path. Dead easy Hey? This Vlisp stuff is easy!!!

Right, we've opened the filing cabinet AutoCAD drawer (the Application Object) and had a look at the characteristics of that drawer. Now we need to delve into the drawer and have a look at one of the document folders stored inside.

_$ (vl-load-com)

_$ (setq acadObject (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00adc088>

Store a reference to the Active Document. (the drawing you fool!)
_$ (setq acadDocument (vla-get-ActiveDocument acadObject))
#<VLA-OBJECT IAcadDocument 00ee0f84>

Double click to select the variable "acadDocument" and add it to the "Watch" Window :
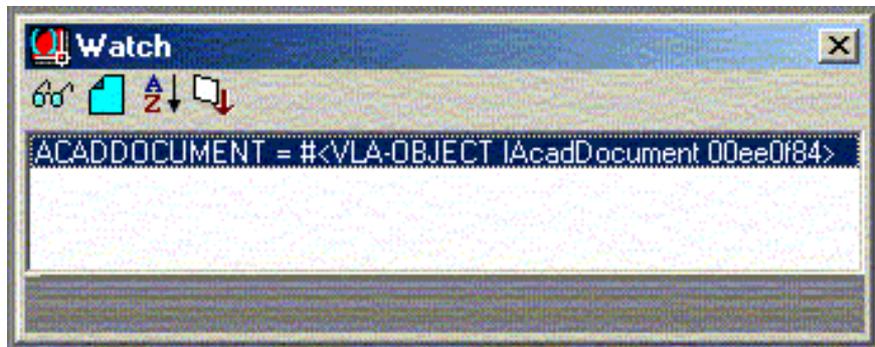
Now double click on "ACADDOCUMENT" Object.
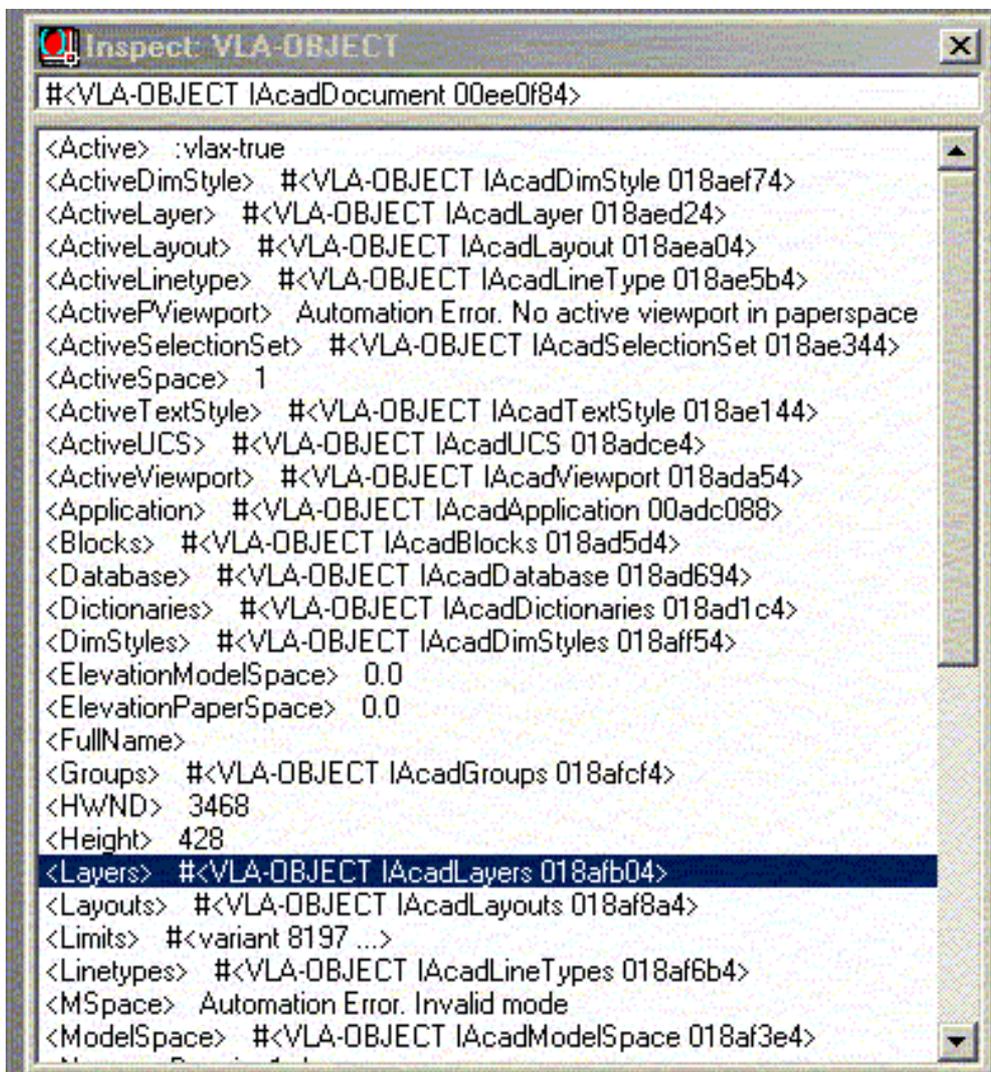
Wow, look at all that! Now double click on <Layers> :
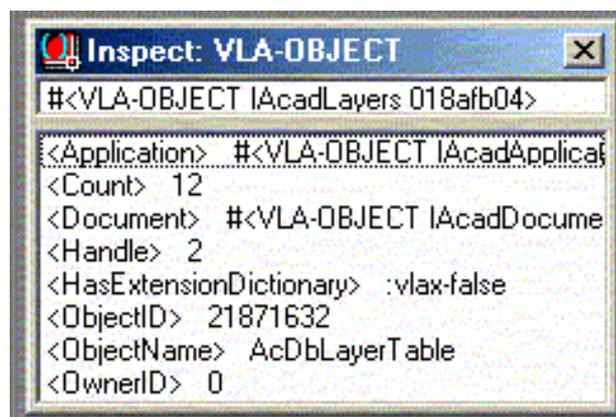
Ha, we've now drilled down to all the layers contained in the drawing. In other words, the "Layers Collection".
"Alright, that's great," I can hear you say, "I can see that there are 13 layers in this drawing, but where are the layers? Right enter this :

_$ (vl-load-com)

_$ (setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))
#<VLA-OBJECT IAcadDocument 00ee0f84>

```
_$ (setq theLayers (vla-get-layers acadDocument))
```
#<VLA-OBJECT IAcadLayers 018aee84>

We are now proud owners of all the layers in this particular drawing. Would you like to have a wee look at them? Just be very careful and don't drop them!
Enter the following at the Console prompt, pressing "Ctrl" then "Enter" after each line.
After the final line, the closing parenthesis, press "Enter" :

```
_$ (setq i 0)
(repeat (vla-get-count theLayers)
(setq aLayer (vla-item theLayers i))
(princ (vla-get-name aLayer))
(terpri)
(setq i (1+ i))
)
```

All the layer names in your drawing should be listed. Mine looks like this :

0
1
2
3
4
5
6
7
8
9
10
12
DEFPOINTS

Would you like to have a closer look at one of the layers in your drawing? Then type this :

```
(setq aLayer (vla-item theLayers "2"))
```
#<VLA-OBJECT IAcadLayer 018aeec4>

We have now accessed Layer "2" from the Layers Collection. Let's list all the properties and methods of this Layer :

```
_$ (vlax-dump-object aLayer T)
```

; IAcadLayer: A logical grouping of data, similar to transparent acetate overlays on a drawing
; Property values:
; Application (RO) = #<VLA-OBJECT IAcadApplication 00adc088>
; Color = 2
; Document (RO) = #<VLA-OBJECT IAcadDocument 00ee0f84>
; Freeze = 0
; Handle (RO) = "40"
; HasExtensionDictionary (RO) = 0
; LayerOn = -1
; Linetype = "DASHED2"
; Lineweight = -3
; Lock = 0
; Name = "2"
; ObjectID (RO) = 21872128

```
; ObjectName (RO) = "AcDbLayerTableRecord"
; OwnerID (RO) = 21871632
; PlotStyleName = "Color_511"
; Plottable = -1
; ViewportDefault = 0
; Methods supported:
; Delete ()
; GetExtensionDictionary ()
; GetXData (3)
; SetXData (2)
T
```

Want to change your Layers color? Enter this :

```
_$ (vla-put-color aLayer 4)
nil
```

This will have changed your Layer "2" color to "Cyan" or color "4", including the color of all Objects within the drawing on Layer "2" with color "ByLayer".

Have a look around at the Document Object. There's stacks to see and drool over. Can you now see the power and capabilities of Visual Lisp?

Next we'll go even deeper and start to have a look at creating, selecting and changing drawing entities.

Before we can create entities, or Objects, within AutoCAD, we need to decide "where" we want to draw, Model Space or Paper Space. And to do that we need to create a reference to the area in which we would like to draw.

Remember our filing cabinet? The Application Object was our drawer, the Document Object was our folder, and now the Model Space Object will be our piece of paper. Let's have a look at the Model Space Object :

_$ (setq modelSpace (vla-get-ModelSpace (vla-get-ActiveDocument (vlax-get-Acad-Object))))
#<VLA-OBJECT IAcadModelSpace 018ade24>

Now run a dump on the modelSpace Object :

```
_$ (vlax-dump-object modelSpace T)
; IAcadModelSpace: A special Block object containing all model space entities
; Property values:
; Application (RO) = #<VLA-OBJECT IAcadApplication 00adc088>
; Count (RO) = 0
; Document (RO) = #<VLA-OBJECT IAcadDocument 00ee0f84>
; Handle (RO) = "18"
; HasExtensionDictionary (RO) = 0
; IsLayout (RO) = -1
; IsXRef (RO) = 0
; Layout (RO) = #<VLA-OBJECT IAcadLayout 018ad934>
; Name = "*MODEL_SPACE"
; ObjectID (RO) = 21871808
; ObjectName (RO) = "AcDbBlockTableRecord"
; Origin = (0.0 0.0 0.0)
; OwnerID (RO) = 21871624
; XRefDatabase (RO) = AutoCAD.Application: No database
; Methods supported:
; Add3DFace (4)
; Add3DMesh (3)
; Add3DPoly (1)
; AddArc (4)
; AddAttribute (6)
; AddBox (4)
; AddCircle (2)
; AddCone (3)
; AddCustomObject (1)
; AddCylinder (3)
; AddDim3PointAngular (4)
; AddDimAligned (3)
; AddDimAngular (4)
; AddDimDiametric (3)
; AddDimOrdinate (3)
; AddDimRadial (3)
; AddDimRotated (4)
; AddEllipse (3)
; AddEllipticalCone (4)
; AddEllipticalCylinder (4)
; AddExtrudedSolid (3)
; AddExtrudedSolidAlongPath (2)
; AddHatch (3)
; AddLeader (3)
; AddLightWeightPolyline (1)
```

```
; AddLine (2)
; AddMInsertBlock (10)
; AddMLine (1)
; AddMText (3)
; AddPoint (1)
; AddPolyfaceMesh (2)
; AddPolyline (1)
; AddRaster (4)
; AddRay (2)
; AddRegion (1)
; AddRevolvedSolid (4)
; AddShape (4)
; AddSolid (4)
; AddSphere (2)
; AddSpline (3)
; AddText (3)
; AddTolerance (3)
; AddTorus (3)
; AddTrace (1)
; AddWedge (4)
; AddXline (2)
; AttachExternalReference (8)
; Bind (1)
; Delete ()
; Detach ()
; GetExtensionDictionary ()
; GetXData (3)
; InsertBlock (6)
; Item (1)
; Reload ()
; SetXData (2)
; Unload ()
T
```

Interesting hey? Now let's draw something. Enter this coding :

```
(setq pt1 (getpoint "\nSpecify First Point : "))
(while (setq pt2 (getpoint "\nSpecify next point : " pt1))
(vla-addline modelSpace (vlax-3d-point pt1) (vlax-3d-point pt2))
(setq pt1 pt2)
)
(428.748 578.851 0.0)
(524.783 509.712 0.0)
```

At last, we've finally drawn something. Now type this :

```
_$ (setq pt1 (getpoint "\nSpecify First Point : "))
(setq pt2 (getpoint "\nSpecify next point : " pt1))
(setq ourLine (vla-addline modelSpace (vlax-3d-point pt1) (vlax-3d-point pt2)))
(922.321 585.542 0.0)
(1016.12 300.064 0.0)
#<VLA-OBJECT IAcadLine 018ab094>
```

Did you notice the *(setq ourLine* statement? This sets a reference to our Line Object. Let's run a dump on that :

```
_$ (vlax-dump-object ourLine T)
; IAcadLine: AutoCAD Line Interface
; Property values:
; Angle (RO) = 5.02985
; Application (RO) = #<VLA-OBJECT IAcadApplication 00adc088>
; Color = 256
; Delta (RO) = (93.8012 -285.479 0.0)
; Document (RO) = #<VLA-OBJECT IAcadDocument 00ee0f84>
; EndPoint = (1016.12 300.064 0.0)
; Handle (RO) = "95D"
; HasExtensionDictionary (RO) = 0
; Hyperlinks (RO) = #<VLA-OBJECT IAcadHyperlinks 018aa2b4>
; Layer = "7"
; Length (RO) = 300.494
; Linetype = "BYLAYER"
; LinetypeScale = 1.0
; Lineweight = -1
; Normal = (0.0 0.0 1.0)
; ObjectID (RO) = 21873192
; ObjectName (RO) = "AcDbLine"
; OwnerID (RO) = 21871808
; PlotStyleName = "ByLayer"
; StartPoint = (922.321 585.542 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 ()
T
```

We now have a list of all the Properties and Methods of our Line Object. Let's change some of it's properties :

Let's change it's Layer:
```
_$ (vla-put-layer ourLine 2)
nil
```

And now it's color :

_$ (vla-put-color ourLine 6)
nil

Let's delete our line :

_$ (entdel (vlax-vla-object->ename ourLine))
<Entity name: 14dc1f8>

**Did you notice how we used a standard AutoLisp command *(entdel)* to delete the line?
But before we could do that, we had to convert the VLisp Object Reference to an AutoLisp
Entity Name.**

**You are probably wondering why we don't just use the AutoLisp (command) function to
draw or change entities. If you use reactor call-back functions, you are not allowed to use
the (command) function.**

**Well, I hope this has given you an insight into the use of Visual Lisp? Don't worry, we'll be
looking a lot closer at all aspects of Visual Lisp in future Tutorials. This is just to get you
started and give you a taste of what's to come.**

Viewing AutoCAD Objects

Visual Lisp has a great way of letting you view AutoCAD Objects by allowing you to "walk" through the drawing database. Let's have a wee look at what's on offer. Open AutoCAD with a blank drawing, and then open the Visual Lisp Editor. Let's set up what information we would like to see first :

- Choose "Tools" - "Environment Options" - "General Options".
- Click the "Diagnostic tab" in the "General Options" window.
- Select "Inspect Drawing Objects Verbosely to view detailed entity information.
- Now choose "View" - "Browse Drawing Database" - "Browse Tables".

The "Drawing Tables" Inspection window will now open :
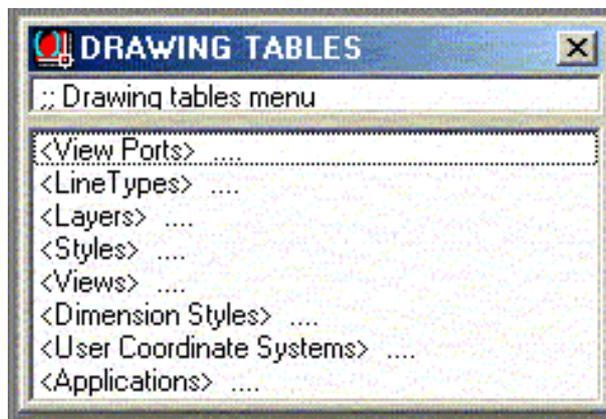
That's very nice!! We have a list of the symbol tables in our drawing.
Now, double-click on <Layers> :
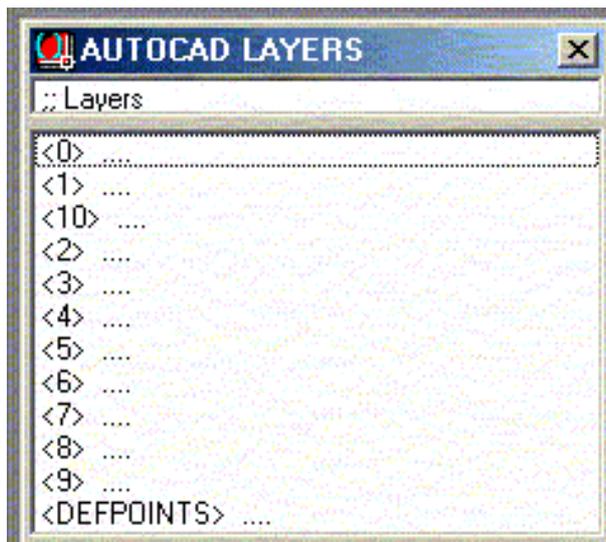
Even better! Now we've got a list of all the Layers in our drawing.
Double-click on Layer <2> :

AUTOCAD TABLE ENTRY

:: LAYER 2

{raw-data} ([0 . "LAYER"] [2 . "2"] [70 . 0] [62 . 2] [6 . "DASHED2"])
{name} "2"
{color} 2
Layer is ON and Thawed
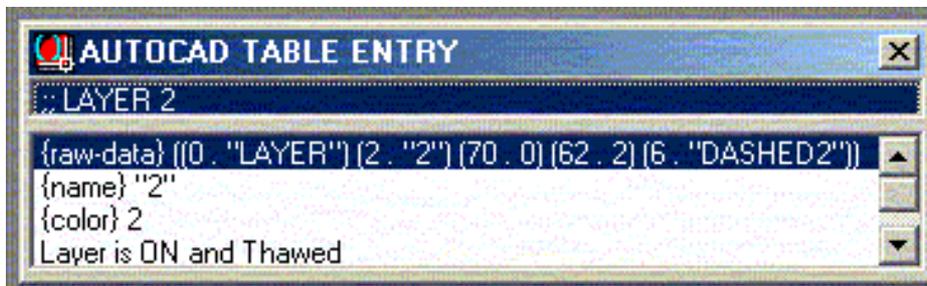
Crikey, now we've got a list of all the Layers attributes "including" the AutoLisp Entity list!!!

Now draw a line, a circle, and some text anywhere in your drawing and then select "View" - "Browse Drawing Database" - "Browse All Entities". The following Inspect window will open :

AUTOCAD ENTITIES

:; AutoCAD Entities list

<CIRCLE> <Entity name: 14db1f8>
<LINE> <Entity name: 14db200>
<TEXT> <Entity name: 14db208>

Now double-click on <TEXT> :

ACAD TEXT

<Entity name: 14db208>

{handle} "959"
{LAYER..} "6"
{extrusion direction} (0.0 0.0 1.0)

{string} "AFRALISP"
{style} "debeers"
{insertion} (504.682 306.755 0.0)
{alignment point} (0.0 0.0 0.0)
{height} 3.5
{relative X-scale} 1.0
{rotation angle} 0.0
{oblique angle} 0.0872665
{adjustment} Left
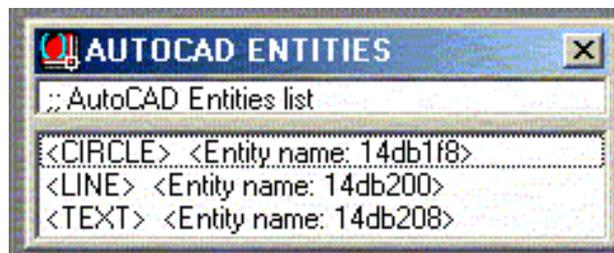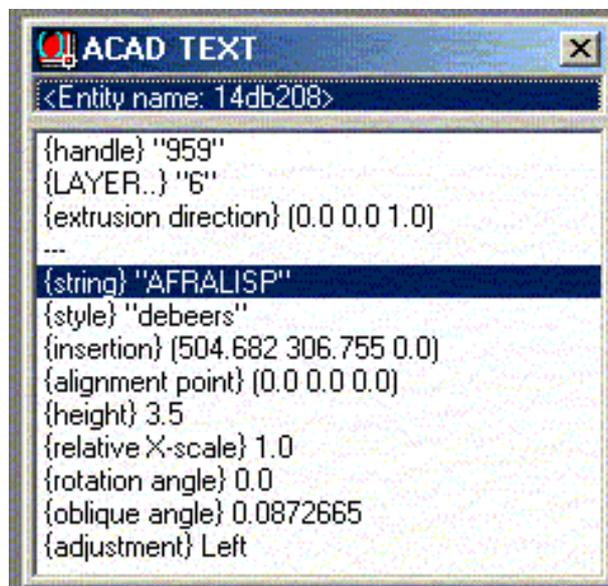
Now we've got a list of all the text attributes.

Just one more so that you get the idea. This time choose "View" - "Browse Drawing Database" - "Browse Selection". The AutoCAD window will appear prompting you to select objects. Do just that :
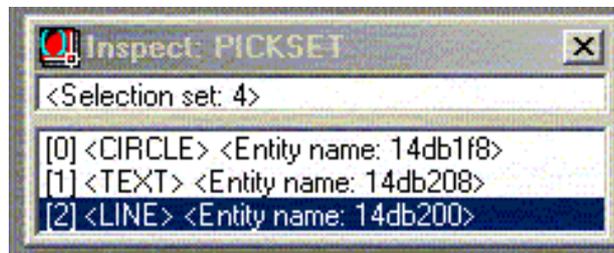
A list of all entities contained in your selection set will be displayed. This time double-click on the <Line> object :
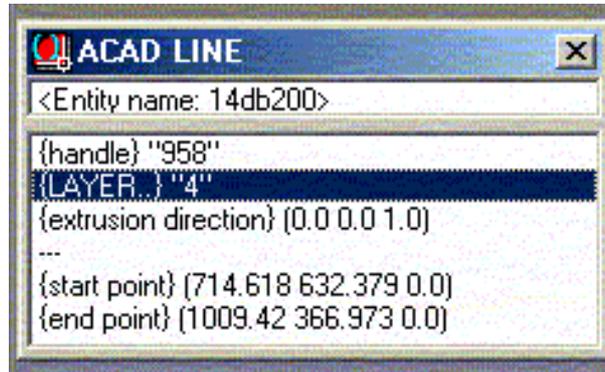
Again a list of the Objects attributes are displayed.
Insert a few more Objects into your drawing such as blocks, blocks with attributes, polylines, hatches,etc.
This set of tools gives you a good idea of how the AutoCAD Object Model is put together. Play around the them, you'll learn a lot........

Properties and Methods

The aim of this tutorial is not to describe the Properties and Methods of every AutoCAD Object, but rather to show you, first how to find the Properties and Methods available for an Object, and secondly to describe the usage of such Property or Method within Visual Lisp. Each Object within AutoCAD has numerous Properties and Methods which differ from Object to Object. To attempt to list each Property or Method for each Object is way beyond the scope of this tutorial.

First of all, let's have a look at defining a Property and a Method.

Visual Lisp Objects support properties and methods, In Visual Lisp, an Object's data (settings or attributes) are called 'properties', while the various procedures that can operate on an Object are called it's 'methods'.

You can change an Object's characteristics by changing it's properties.
Consider a radio: One property of a radio is its volume. In Visual Lisp,
you might say that a radio has a 'Volume' property that you can adjust by
changing its value. Assume you can set the volume of a radio from 0 to 10.
If you could control a radio with Visual Lisp, you might write code in a
procedure that changes the value of the 'Volume' property from 3 to 5 to
make it play louder :

(vla-put-Volume Radio 5)

In addition to properties, objects have methods. Methods are part of objects
just as properties are. Generally, methods are actions you want to perform,
while properties are the attributes you set or retrieve. For example, you dial
a telephone to make a call. You might say that telephones have a 'Dial' method, and you could
use this syntax to dial a seven digit number 3334444:

(vla-Dial Telephone 3334444)

Properties

Angle
Application
Color
Document
Delta
EndPoint
Handle
HasExtensionDictionary
Hyperlinks
Layer
Length
Linetype
LinetypeScale
Lineweight
Normal
ObjectID
OwnerID
PlotStyleName
StartPoint
Thickness

Before you can start to change an Objects Properties or Methods, you need to know what Properties and Methods are available to the particular Object. There are a couple of ways of going about this. First we'll look at Properties.

Under AutoCAD Help, open "Visual Lisp and AutoLisp" and then "ActiveX and VBA Reference". Choose the "Objects" sub-section and from the list choose the Object whose Properties you would like list. Choose "Line".

As you can see, all the Properties applicable to the "Line" Object are listed.

Be careful though, as some of these Properties are "*Read Only*" and cannot be changed.
e.g. The "Angle" property is "Read Only." Think about it, if you changed the "ANGLE" Property, the Start or End point of the Line Object would have to change as well.

Click on any of the Property hyperlinks for further information.

Another way of finding an Objects properties is to use the Visual Lisp function *(vlax-dump-object)*.

Open AutoCAD and then the Visual Lisp editor and type the following at the Console prompt :

```
_$ (vl-load-com)

_$ (setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))
#<VLA-OBJECT IAcadDocument 00b94e14>

_$ (setq mspace (vla-get-modelspace acadDocument))
#<VLA-OBJECT IAcadModelSpace 01e42494>

_$ (setq apt (getpoint "Specify First Point: "))
(228.279 430.843 0.0)

_$ (setq pt (getpoint "Specify next point: " apt))
(503.866 538.358 0.0)

_$ (setq myline (vla-addline mspace (vlax-3d-point apt)(vlax-3d-point pt)))
#<VLA-OBJECT IAcadLine 01e84614>

_$ (vlax-dump-object myline)
; IAcadLine: AutoCAD Line Interface
; Property values:
;   Angle (RO) = 0.371971
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00adc088>
;   Color = 256
;   Delta (RO) = (275.587 107.515 0.0)
;   Document (RO) = #<VLA-OBJECT IAcadDocument 00b94e14>
```

```
;   EndPoint = (503.866 538.358 0.0)
;   Handle (RO) = "958"
;   HasExtensionDictionary (RO) = 0
;   Hyperlinks (RO) = #<VLA-OBJECT IAcadHyperlinks 01e84564>
;   Layer = "7"
;   Length (RO) = 295.817
;   Linetype = "BYLAYER"
;   LinetypeScale = 1.0
;   Lineweight = -1
;   Normal = (0.0 0.0 1.0)
;   ObjectID (RO) = 25187840
;   ObjectName (RO) = "AcDbLine"
;   OwnerID (RO) = 25186496
;   PlotStyleName = "ByLayer"
;   StartPoint = (228.279 430.843 0.0)
;   Thickness = 0.0
;   Visible = -1
```

The (vlax-dump object) function lists all the available properties for a particular Object.
Note the (*RO*) after some of the Properties. This tells you that this Property is "*Read Only*".

Methods

ArrayPolar
ArrayRectangular
Copy
Delete
GetBoundingBox
GetExtensionDictionary
GetXData
Highlight
IntersectWith
Mirror
Mirror3D
Move
Offset
Rotate
Rotate3D
ScaleEntity
SetXData
TransformBy
Update

Let's have a look at the Methods pertaining to an Object.
Under AutoCAD help, open "Visual Lisp and AutoLisp" and then "ActiveX and VBA Reference". Again, choose the "Objects" sub-section and from the list choose the Object whose Methods you would like list. Choose "Line".
As you can see, all the Methods applicable to the "Line" Object are listed.

Click on the "Move" method.
The VBA Method for "Move" is displayed and the syntax is as follows :

object.Move Point1, Point2

Object : *All Drawing Objects, AttributeRef. The object or objects this method applies to.*

Point1 : *Variant (three-element array of doubles); input-only. The 3D WCS coordinates specifying the first point of the move vector.*

Point2 : *Variant (three-element array of doubles); input-only. The 3D WCS coordinates specifying the second point of the move vector.*

Let's move the line we've just drawn :

```
_$ (setq apt2 (getpoint "Specify Base Point: "))
(220.911 526.575 0.0)

_$ (setq pt2 (getpoint "Specify second point: " apt2))
(383.02 617.889 0.0)
```

```
_$ (vla-move myline (vlax-3d-point apt2)(vlax-3d-point pt2))
```

Now let's "dump" the Object, but this time we'll use the "*T*" flag to display the Objects
Methods as well as it's Properties.

```
_$ (vlax-dump-object myline T)
; IAcadLine: AutoCAD Line Interface
; Property values:
;    Angle (RO) = 5.60729
;    Application (RO) = #<VLA-OBJECT IAcadApplication 00adc088>
;    Color = 256
;    Delta (RO) = (246.112 -197.356 0.0)
;    Document (RO) = #<VLA-OBJECT IAcadDocument 00b94e14>
;    EndPoint = (629.133 420.533 0.0)
;    Handle (RO) = "957"
;    HasExtensionDictionary (RO) = 0
;    Hyperlinks (RO) = #<VLA-OBJECT IAcadHyperlinks 01e84574>
;    Layer = "7"
;    Length (RO) = 315.469
;    Linetype = "BYLAYER"
;    LinetypeScale = 1.0
;    Lineweight = -1
;    Normal = (0.0 0.0 1.0)
;    ObjectID (RO) = 25187832
;    ObjectName (RO) = "AcDbLine"
;    OwnerID (RO) = 25186496
;    PlotStyleName = "ByLayer"
;    StartPoint = (383.02 617.889 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 ()

"But, the syntax of the "Move" method you used in Visual Lisp, is different from the VBA syntax!! How do I know how to call the function in Visual Lisp?"

Don't worry, we'll be having a look at that on the next page.

Before we go there, here's a little application that will dump all Properties and Methods for selected Objects :

;coding starts here

```
(defun C:HaveaDump ( / ent)

  (vl-load-com)

    (while

      (setq ent (entsel))

      (vlax-dump-object (vlax-Ename->Vla-Object (car ent)) T)

    );while

    (princ)

);defun

(princ)
```

;coding ends here

On the next page we'll have a look at how we call Property and Method Functions. See you there.......

How to Call a Visual Lisp Function

Right, you've identified the Visual Lisp Property or Method that you need, but you still need to determine how to call the function. You need to know the arguments to specify and the data type of those arguments.
Let's look at Properties first.

The syntax for the Layer Property in VBA is as follows :

object.Layer or *object.property*

object : *All Drawing objects, AttributeRef, Group. The object or objects this property applies to.*
Layer : *String; read-write (write-only for the Group object). The name of the layer.*

Remarks

All entities have an associated layer. The document always contains at least one layer (layer 0). As with linetypes, you can specify a layer for an entity. If you don't specify a layer, the current active layer is used for a new entity. If a layer is specified for an entity, the current active layer is ignored. Use the ActiveLayer property to set or query the current active layer.
Each layer has associated properties that can be set and queried through the Layer object.

In VBA you would use

Oldlayer = object.Layer to retrieve the Layer Name, and

object.Layer = "2" to change the Layer Name.

Visual Lisp provides functions for reading and updating Object Properties.
Functions that read Object Properties are named with a *vla-get* prefix and require the following syntax :

*(*vla-get-*property object)*

For example, "*vla-get-layer object*" returns the Layer the Object is on.

Enter this in the Visual Lisp Console :

_$ (vl-load-com)

_$ (setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))
#<VLA-OBJECT IAcadDocument 00b94e14>

_$ (setq mspace (vla-get-modelspace acadDocument))
#<VLA-OBJECT IAcadModelSpace 01e42444>

_$ (setq apt (getpoint "Specify First Point: "))
(307.86 539.809 0.0)

_$ (setq pt (getpoint "Specify next point: " apt))

_$ (setq myline (vla-addline mspace (vlax-3d-point apt)(vlax-3d-point pt)))
#<VLA-OBJECT IAcadLine 01e81de4>

_$ (setq oldlayer (vla-get-layer myline))
"7"

The variable "*oldLayer*" now contains the Layer name of your Line Object.

Functions that update Properties are prefixed with *vla-put* and use the following syntax :

(vla-put-*property object new-value*)

For example, "*vla-put-layer object new-value*" changes the layer of the Object.

Enter this at the Console prompt :

_$ (vla-put-layer myline "4")
nil

Your line will have now changed to Layer "4".

Let's have a look at Methods now.
The syntax for the "Offset" Method in VBA is as follows :

RetVal = object.Offset(Distance) or Return Value = object.Method (arguments)

Object : Arc, Circle, Ellipse, Line, LightweightPolyline, Polyline, Spline, XLine. The object or objects this method applies to.
Distance : Double; input-only. The distance to offset the object. The offset can be a positive or negative number, but it cannot equal zero. If the offset is negative, this is interpreted as being an offset to make a "smaller" curve (that is, for an arc it would offset to a radius that is "Distance less" than the starting curve's radius). If "smaller" has no meaning, then it would offset in the direction of smaller X, Y, and Z WCS coordinates.

RetVal : Variant (array of objects). An array of the newly created objects resulting from the offset.

In VBA you would use

offsetObj = object.offset(15.5) to Offset an Object.

The syntax definitions used in the "ActiveX and VBA Reference" were designed for Visual Basic Users. Consider the VBA Offset Method :

returnvalue = object.Method (arguments)

returnvalue = object.Offset(Distance) or using the names in our example

offLine = myline.Offset(15.5)

The syntax for the same operation in Visual Lisp is :

(setq returnvalue (vla-method object argument)) or using our names

(setq offLine (vla-offset myline 15.5))

Different Objects have different Methods but the same principle applies.

Type this at the Console prompt :

_$ (setq offLine (vla-offset myline 15.5))
#<variant 8201 ...>

The variable "*offLine*", now contains the data for your newly created line in the form of a variant array.

So to recap :

- *vla-get-* functions correspond to every ActiveX Property, enabling you to retrieve the value of that Property (for example, *vla-get-Color* obtains an Object's Color Property.
- *vla-put-* functions correspond to every Property, enabling you to update the value of the Property (for example, *vla-put-Color* updates an Objects Color Property.
- *vla-* functions correspond to every ActiveX Method. Use these functions to invoke the Method (for example, *vla-addCircle* invokes the *addCircle* Method.

Visual Lisp also adds a set of ActiveX-related functions whose names are prefixed with *vlax-*. These are more general ActiveX functions, each of which can be applied to numerous Methods, Properties or Objects. For example, with the *vlax-get-property* function, you can obtain any Property of any ActiveX Object.

On the next page we'll have a look at how we can determine whether an Object is available for updating and whether a Method or a Property applies to an Object.

Determining Whether an Object is Available for Updating

If other applications are working with any AutoCAD Objects at the same time as your program, those Objects may not be accessible. This is especially important to lookout for if your application includes reactors, because reactors execute code segments in response to external events that cannot be predicted in advance. Even a simple thing such as a locked Layer can prevent you from changing an Objects Properties.

Visual Lisp provides the following functions to test the accessibility of an Object before trying to use the Object :

- *vlax-read-enabled-p* Tests whether you can read an Object.
- *vlax-write-enabled-p* Determines whether you can modify an Objects Properties.
- *vlax-erased-p* Checks to see if an Object has been erased. Erased Objects may still exist in the drawing database.

All these test functions return *T* if true, and *nil* if false.
Let's test them out. Draw a line anywhere in AutoCAD then enter this at the Console prompt :

_$ (vl-load-com)

_$ (setq ent (entsel))
(<Entity name: 17e39f8> (434.601 389.588 0.0))

_$ (setq myLine (vlax-Ename->Vla-Object (car ent)))
#<VLA-OBJECT IAcadLine 01e81f84>

Determine whether the line is readable :

_$ (vlax-read-enabled-p myLine)
T

Determine whether the line is modifiable :

_$ (vlax-write-enabled-p myLine)
T

See if the line has been erased :

_$ (vlax-erased-p myLine)
nil

Erase the line :

_$ (vla-delete myLine)
nil

See if the line is still readable :

_$ (vlax-read-enabled-p myLine)
nil

Check to confirm that the object has been deleted :

_$ (vlax-erased-p myLine)
T

Determining If a Method or Property Applies to an Object

Trying to use a Method that does not apply to a specified Object will result in an error. The same goes for trying to reference a Property that does not apply to an Object. This will also result in an error. In instances where you are not sure what applies, use the *vlax-method-applicable-p* and the *vlax-property-available-p* functions.
These functions return *T* if the Method or Property is available for the Object, and *nil* if not.

The syntax for *vlax-method-applicable-p* is :

(vlax-method-applicable-p *object method*)

The following command checks to see if the "Copy" Method can be applied to an Object :

_$ (vlax-method-applicable-p *myLine "Copy"*)
T

The following command determines if the "addBox" Method can be applied to an Object :

_$ (vlax-method-applicable-p *myLine "addBox"*)
nil

For *vlax-property-available-p* the syntax is :

(vlax-property-available-p object property [T])

The following commands determine if Color and Center are properties of the myLine Object :

(vlax-property-available-p myLine "Color")
T
(vlax-property-available-p myLine "Center")
nil

Supplying the optional "*T*" argument to *vlax-property-available-p* changes the meaning of the test. If you supply this argument, the function returns *T* only if the Object has the Property AND the Property can be modified. If the Object has no such Property or the Property is *Read-Only*, *vlax-property-available-p* returns nil.

For example, a Circle contains an Area Property, but you cannot update it.
If you check the Property without specifying the optional argument the result is *T*.

(vlax-property-available-p myCircle "area")
T

If you supply the optional argument, the result is nil :

(vlax-property-available-p myCircle "area" T)
nil

Well, that's it with Methods and Properties. Time for me to pour myself a nice long, cold beer and retire to the garden and sit in the sun. The advantages of living in Africa. Eat your heart out............

Oh, by the way, my Mum has got a great Method of cooking Tripe and Onion's if anybody is interested!!

Arrays

If you've programmed in VBA or other languages, you're probably familiar with the concept of arrays. An array is a named collection of variables of the same data type. Each array element can be distinguished from other elements by one or more integer indexes. For example, if the "*sheetNames*" array contains three names, you can set and return the names in VBA as shown in the following example :

```
sheetNames(0) = "Sheet1"
sheetNames(1) = "Sheet2"
sheetNames(2) = "Sheet3"

MsgBox sheetNames(1)
```

Would return a message :

"Sheet2"

Arrays allow you to group related variables in a way that makes it easier for you to keep track of, access, and manipulate them all at once, while still being able to access each variable individually. This helps you create smaller and simpler routines in many situations, because you can set up loops using index numbers to deal efficiently with any number of cases.

When you create an array, its size is determined by the number of dimensions it has and the by the upper and lower bounds for the index numbers in each dimension. The "*sheetNames*" array in the earlier example has one dimension and three elements; the lower bound is 0 and the upper bound is 2.

"That's fine Kenny, but how do we create an Array in Visual Lisp?"
O.K. I hear you, there's no need to yell!
Enter this at the Console prompt :

```
_$ (vl-load-com)

_$ (setq sheet_type (vlax-make-safearray vlax-vbString '(0 . 2)))
#<safearray...>

_$ (vlax-safearray-fill sheet_type '("Sheet1" "Sheet2" "Sheet3"))
#<safearray...>
```

Have a look at the "*sheet_ type*" variable in the Watch window :
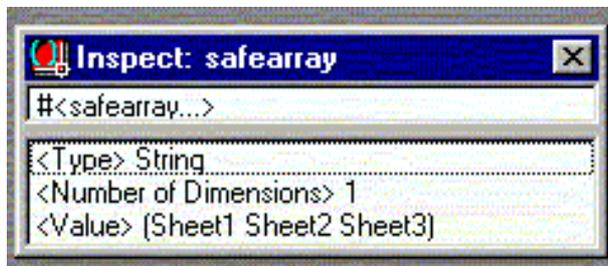
This tells us that the Array contains strings, has one dimension and contains 3 elements. Let's convert it to an AutoLisp List :

```
_$ (setq alist (vlax-safearray->list sheet_type))
("Sheet1" "Sheet2" "Sheet3")
```

Did a light just go off in your head? An Array is just a List in a slightly different format. To create an Array, or safearray as they are know in Visual Lisp, we use the "*vlax-make-safearray*" function. To populate a safearray we use the "*vlax-safearray-fill*" or the "*vlax-safearray-put*" functions.

The "*vlax-make-safearray*" function requires a minimum of two arguments. The first argument identifies the type of data that will be stored in the array.
One of the following constants must be specified for the data type :

vlax-vbInteger	Integer
vlax-vbLong	Long Integer
vlax-vbSingle	Single-precision floating point number
vlax-vbDouble	Double-precision floating point number
vlax-vbString	String
vlax-vbObject	Object
vlax-vbBoolean	Boolean
vlax-vbVariant	Variant

The remaining arguments to "*vlax-make-safearray*" specify the upper and lower bounds of each dimension of the array. The lower bound for an index can be zero or any positive or negative number. Have another look at the function we called earlier :

```
_$ (setq sheet_type (vlax-make-safearray vlax-vbString '(0 . 2)))
```

This function created a single-dimension array consisting of three strings with a starting index of 0 (element 0, element 1 and element 2).
Consider this :

```
_$ (setq pt1 (vlax-make-safearray vlax-vbDouble '(1 . 3)))
```

The lower bound specified in this example is one and the upper bound specified is three,

so the array will hold three doubles (element 1, element 2 and element 3).

The "*vla-safearray-fill*" function requires two arguments: the variable containing the array you are populating and a list of the values to be assigned to the array elements. You must specify as many values as there are elements in the array or *vla-safearray-fill*" results in an error.
The following code populates a single-dimension array of three doubles:

```
_$ (vlax-safearray-fill pt1 '(100 100 0))
```

To convert an array to an AutoLisp list, you can use the *(vlax-safearray->list)* function. Try it out :

```
_$ (vlax-safearray->list pt1)
(100.0 100.0 0.0)
```

Let's create a Array with two dimensions, each dimension with three elements:

```
_$ (setq two_dim (vlax-make-safearray vlax-vbString '(0 . 1) '(1 . 3)))
#<safearray...>

_$ (vlax-safearray-fill two_dim '(("Sheet1" "Sheet2" "Sheet3") ("a" "b" "c")))
#<safearray...>

_$ (vlax-safearray->list two_dim)
(("Sheet1" "Sheet2" "Sheet3") ("a" "b" "c"))
```

This is just a list of lists.
The first list, *'(0 .1)* is the number of dimensions.
The second list, *'(1 . 3)* is the number of elements

And now a three dimensional Array with two elements in each dimension:

```
_$ (setq three_dim (vlax-make-safearray vlax-vbString '(0 . 2) '(1 . 2)))
#<safearray...>

_$ (vlax-safearray-fill three_dim '(("Sheet1" "Sheet2") ("a" "b") ("d" "e")))
#<safearray...>

_$ (vlax-safearray->list three_dim)
(("Sheet1" "Sheet2") ("a" "b") ("d" "e"))
```

Here we have a list of three lists.
This time, the first list *'(0 . 2)* defines three dimensions and the second *'(1 . 2)* defines 2 elements in each dimension.

One place you will be using "*vlax-safearray-fill*" is when creating selection sets with filters. The syntax in ActiveX for "Selecting All with Filters" is as follows "

object.Select Mode[, Point1][, Point2][, Filter_Code][, Filter_Value]

Filter_Code must be an Integer array and **Filter_Value** a Variant array.
In Visual Lisp, the coding would be written like this :

```
;create a 2 element integer array for the DXF Codes.
(setq filter_code (vlax-make-safearray vlax-vbinteger '(0 . 1)))

;create a 2 element variant array for the values.
(setq filter_value (vlax-make-safearray vlax-vbvariant '(0 . 1)))

;DXF Codes for Objects and Layer : " 0" for Object," 8" for Layer.
(vlax-safearray-fill filter_code '(0 8))

;Name of Object and Layer.
(vlax-safearray-fill filter_value '("CIRCLE" "2"))

;select ALL Circles on Layer 2.
(vla-select newsset acSelectionSetAll nil nil filter_code filter_value)
```

For more information on Selections Sets, pop along to the "Selection Sets" tutorial section and get yourself even more confused.

The "*vlax-safearray-put-element*" function can be used to assign values to one or more elements of a safearray. The number of arguments required by this function depends on the number of dimensions in the array.

- ■ The first argument always names the safearray to which you are assigning a value.
- ■ The next set of arguments identifies index values pointing to the element to which you are assigning a value. For a single-dimension array, specify one index value: for a two-dimension array, specify two index values, and so on.
- ■ The final argument is always the value to be assigned to the safearray element.

Have a look at the following :

```
_$ (setq pt1 (vlax-make-safearray vlax-vbDouble '(1 . 3)))
#<safearray...>

_$ (vlax-safearray-put-element pt1 1 100)
100

_$ (vlax-safearray-put-element pt1 2 100)
100

_$ (vlax-safearray-put-element pt1 3 75)
75

_$ (vlax-safearray->list pt1)
```

(100.0 100.0 75.0)

_$ (vlax-safearray-put-element pt1 1 50)
50

_$ (vlax-safearray->list pt1)
(50.0 100.0 75.0)

Now let's populate a two-dimension array of strings :

_$ (setq two_dim (vlax-make-safearray vlax-vbString '(0 . 1) '(1 . 3)))
#<safearray...>

_$ (vlax-safearray-put-element two_dim 0 1 "a")
"a"

_$ (vlax-safearray->list two_dim)
(("a" "" "") ("" "" ""))

_$ (vlax-safearray-put-element two_dim 0 2 "b")
"b"

_$ (vlax-safearray-put-element two_dim 0 3 "c")
"c"

_$ (vlax-safearray-put-element two_dim 1 1 "d")
"d"

_$ (vlax-safearray-put-element two_dim 1 2 "e")
"e"

_$ (vlax-safearray-put-element two_dim 1 3 "f")
"f"

_$ (vlax-safearray->list two_dim)
(("a" "b" "c") ("d" "e" "f"))

You can use "*vlax-safearray-get-element*" to get the value of any element in any array. Here we'll use "*vlax-safearray-get-element*" to retrieve the second element in the first dimension of the array:

_$ (vlax-safearray-get-element matrix 1 2)
"b"

(*vlax-safearray-get-l-bound*) returns the lower boundary (starting index) of a dimension of an array :

Get the starting index value of the second dimension of the array:

```
_$ (vlax-safearray-get-l-bound two_dim  2)
```
1

The second dimension starts with index 1.

Conversley, "*vlax-safearray-get-u-bound*" returns the upper boundary (end index) of a dimension of an array

Get the end index value of the second dimension of the array:

```
_$ (vlax-safearray-get-u-bound two_dim  2)
```
3

The second dimension ends with index 3.

You can use "*vlax-safearray-get-dim*" to get the number of dimensions in a safearray object :

Get the number of dimensions in "*two_dim*":

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

There are 2 dimensions in "*two_dim*".

Let's have a look at putting some of this to good use :

```
(vl-load-com)

(defun c:Line_VL ( / acApp acDoc mspace p1 p2 sp ep lineObj)

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

  (setq acDoc (vla-get-activedocument acApp))

  (setq mspace (vla-get-modelspace acDoc))

  (setq p1 (getpoint "\nFirst Point : "))

  (setq p2 (getpoint p1 "\nSecond Point : "))

  (setq sp (vlax-make-safearray vlax-vbdouble '(0 . 2)))

  (setq ep (vlax-make-safearray vlax-vbdouble '(0 . 2)))

  (vlax-safearray-fill sp p1)
```

```
    (vlax-safearray-fill ep p2)

  (setq lineObj (vla-addline mspace sp ep))

 (princ)

);defun
```

There is an easier way of writing this routine :

```
(vl-load-com)
(defun c:Line_VL ( / acApp acDoc mspace p1 p2 lineObj)
    (setq acApp (vlax-get-acad-object))
    (setq acDoc (vla-get-activedocument acApp))
    (setq mspace (vla-get-modelspace acDoc))
    (setq p1 (getpoint "\nFirst Point : "))
    (setq p2 (getpoint p1 "\nSecond Point : "))
    (setq lineObj (vla-addline mspace (vlax-3d-point p1) (vlax-3d-point p2)))
  (princ)
);defun
```

For methods that require you to pass a three-element array of doubles (typically to specify a point), you can use the "*vlax-3d-point*" function.

Well, that's it with Arrays. I haven't covered absolutely everything pertaining to Arrays, but you should now have enough to get you started. I hope that you understood everything I was warbling on about, and that I didn't confuse you too much!!!! Adios for now, amigo........

Selection Objects

Selecting Objects and creating Selection Sets is much the same in Visual
Lisp as it is for standard AutoLisp except for two main differences. All Entities
contained in an AutoLisp selection set, must be converted to VLA Objects
before VLA functions can be applied to them, and you cannot use AutoCAD interactive functions
such as *(entsel)* or *(ssget)* within a reactor callback function.

In this tutorial, we we look at Selecting Objects and creating Selection sets using the *(entsel)*
function, the *(ssget)* function and then using only VLA functions.

Let's look at selecting a single entity first using *(entsel)*.
Consider this coding :

```
(defun selectionlisp1 ( / sset check)

;load the visual lisp extensions
(vl-load-com)

;check for selection
(while

    ;get the entity and entity name
    (setq sset (car (entsel)))

    ;convert to vl object
    (setq sset (vlax-ename->vla-object sset))

    ;check if the entity has a color property
    ;and it can be updated
    (setq check (vlax-property-available-p sset "Color" T))

    ;if it can
    (if check

        ;change it's color
        (vlax-put-property sset 'Color 4)

    );if

);while

  (princ)

);defun

(princ)
```

We select the entity using the standard *(entsel)* function. We then have to
convert the entity to a VLA Object by using the *(vlax-ename->vla-object)*
function.
Next we check to see if the object first has a color property, and secondly
is updateable using the *(vlax-property-available-p)* function with it's "*T*"
argument set.

Finally we change it's color property using the *(vlax-put-property)* function.
Dead easy, hey?

Create Selection sets utilising *(ssget)* is also quite straightforward.
Have a look at this :

```
(defun selectionlisp2 ( / sset item ctr check)

;load the visual lisp extensions
(vl-load-com)

;check for selection
(while

    ;get the selection set
    (setq sset (ssget))

    ;set up the counter
    (setq ctr 0)

    ;count the number of entities and loop
    (repeat (sslength sset)

        ;extract the entity name
        (setq item (ssname sset ctr))

        ;convert to vl object
        (setq item (vlax-ename->vla-object item))

        ;check if the entity has a color property
        ;and it can be updated
        (setq check (vlax-property-available-p item "Color" T))

        ;if it can
        (if check

            ;change it's color
            (vlax-put-property item 'Color 6)

        );if

        ;increment the counter
        (setq ctr (1+ ctr))

    );repeat

);while

  (princ)

);defun

(princ)
```

The only difference here, is that we need to loop through each individual item in the selection set, converting it to a VLA Object, and then checking and changing it's properties.
Again this is standard AutoLisp with a little bit of VLA thrown in to confuse you.

To create a selection set using only VLA is a slightly different matter.
Here we need to access the Object Model to reference, create and store our selection set. Have a look at this :

```
(defun selectionvl ( / ssets acadDocument newsset ctr item)

;load the visual lisp extensions
(vl-load-com)

;retrieve a reference to the documents object
(setq acadDocument (vla-get-activedocument
                (vlax-get-acad-object)))

;retrieve a reference to the selection sets object
(setq ssets (vla-get-selectionsets acadDocument))

;add a new selection set
(setq newsset (vla-add ssets "SS1"))

;select your new selection set objects
(vla-selectOnScreen newsset)

;set the counter to zero
(setq ctr 0)

;count the number of objects and loop
(repeat (vla-get-count newsset)

    ;retrieve each object
    (setq item (vla-item newsset ctr))

    ;check if the entity has a color property
    ;and it can be updated
    (setq check (vlax-property-available-p item "Color" T))

    ;if it can
    (if check

        ;change it's color
        (vlax-put-property item 'Color 5)

    );if

    ;increment the counter
    (setq ctr (1+ ctr))
```

```
  );repeat

  ;delete the selection set
  (vla-delete (vla-item ssets "SS1"))

  (princ)

);defun

(princ)
```

First we need to reference the selection set collection. Looking at the
Object Model, we find this collection to be part of the "*Active Document*" Object.
After grabbing and storing our reference, using the *(vla-get-selectionsets)* function, we then
need to create a new selection set. *(SS1)* using the
(vla-add) function.
O.K. that's done, now what next?
Well, we've got a selection set but there's nothing in it. Using the *(vla-selectOnScreen)* method,
we populate our selection set with the
Objects that we would like to process. Again, we use the *(repeat)* function
to loop through all of our objects, checking if each object has a color
property and is updateable before changing it's color. Of course, we
don't need to convert these Objects as they are already VLA Objects.
The final step in our application is to delete the selection set to prevent
"*Selection Set Already Exists*" errors when we run the routine again.

Selecting with Filters

Now things get a wee bit tricky. I suggest you read the chapter on "Arrays" before you carry on
with this section, as a good understanding of them will help you a lot.

First of all, let's have a look at the VBA syntax for Selecting with Filters :

object.SelectOnScreen [FilterType][, FilterData]

Object : SelectionSet
The object or objects this method applies to.

FilterType : Integer; input-only; optional
A DXF group code specifying the type of filter to use.

FilterData : Variant; input-only; optional
The value to filter on.

ActiveX requires a Filter Type to be an Array of Integers, and the Filter Data to be an Array of
Variants. To Filter for all Objects on Layer "7" we would write it like this in Visual Lisp:

```
;create a single element array for the DXF Code
(setq filter_code (vlax-make-safearray vlax-vbinteger '(0 . 0)))

;create a single element array for the value
(setq filter_value (vlax-make-safearray vlax-vbvariant '(0 . 0)))

;DXF Code for layers
```

```
(vlax-safearray-fill filter_code '(8))

;the filter value
(vlax-safearray-fill filter_value '("7"))

;Use Select on Screen to select objects on Layer 7
(vla-selectOnScreen newsset filter_code filter_value)
```

In AutoLisp you would write this :

```
(setq newsset (ssget '((8 . "7"))))
```

This is how the revised program would look :

```
(defun selectionvl ( / ssets acadDocument newsset ctr item filter_code filter_value)

;load the visual lisp extensions
(vl-load-com)

;retrieve a reference to the documents object
(setq acadDocument (vla-get-activedocument
    (vlax-get-acad-object)))

;retrieve a reference to the selection sets object
(setq ssets (vla-get-selectionsets acadDocument))

;add a new selection set
(setq newsset (vla-add ssets "SS1"))

;create a single element array for the DXF Code
(setq filter_code (vlax-make-safearray vlax-vbinteger '(0 . 0)))

;create a single element array for the value
(setq filter_value (vlax-make-safearray vlax-vbvariant '(0 . 0)))

;DXF Code for layers
(vlax-safearray-fill filter_code '(8))

;the filter value
(vlax-safearray-fill filter_value '("7"))

;Use Select on Screen to select objects on Layer 7
(vla-selectOnScreen newsset filter_code filter_value)

;set the counter to zero
(setq ctr 0)

 ;count the number of objects and loop
 (repeat (vla-get-count newsset)

  ;retrieve each object
  (setq item (vla-item newsset ctr))

  ;check if the entity has a color property
  ;and it can be updated
  (setq check (vlax-property-available-p item "Color" T))

  ;if it can
  (if check
```

```
    ;change it's color
    (vlax-put-property item 'Color 5)

    );if

    ;increment the counter
    (setq ctr (1+ ctr))

  );repeat

  ;delete the selection set
  (vla-delete (vla-item ssets "SS1"))

  (princ)

);defun

(princ)
```

To select ALL the Circles on Layer "2", We would write this :

```
 ;create a 2 element array for the DXF Code
(setq filter_code (vlax-make-safearray vlax-vbinteger '(0 . 1)))

;create a 2 element array for the values
(setq filter_value (vlax-make-safearray vlax-vbvariant '(0 . 1)))

;DXF Code for Objects and Layer
(vlax-safearray-fill filter_code '(0 8))

;the filter values
(vlax-safearray-fill filter_value '("CIRCLE" "2"))

;select ALL Circles on Layer 2
(vla-select newsset acSelectionSetAll nil nil filter_code filter_value)
```

This is the equivalent in AutoLisp :

```
(setq newsset (ssget "x" '((0 . "CIRCLE") (8 . "2"))))
```

To select ALL Circles OR Text, we would write this :

```
;create a 4 element array for the DXF codes
(setq filter_code (vlax-make-safearray vlax-vbinteger '(0 . 3)))

;create a 4 element array for the names
(setq filter_value (vlax-make-safearray vlax-vbvariant '(0 . 3)))

;DXF Codes for OR and Objects
(vlax-safearray-fill filter_code '(-4 0 0 -4))

;the filter values
(vlax-safearray-fill filter_value '("<or" "CIRCLE" "TEXT" "or>"))

;select ALL Circles OR Text
(vla-select newsset acSelectionSetAll nil nil filter_code filter_value)
```

In AutoLisp, this is equivalent to :

(setq newsset (ssget "x" '((-4 . "<or") (0 . "CIRCLE") (0 . "TEXT") (-4 . "or>"))))

Can you see what we are doing?

We are basically "splitting" the AutoLisp dotted pairs into two arrays, one containing the DXF codes, and the other containing the filter values.

There is another way of filtering the Selection Set itself. Consider this code:

(if (= (vlax-get-property item 'ObjectName) "AcDbCircle")

    (vlax-put-property item 'Color 3)

);if

Makes you think doesn't it?

Selection Set Already Exists

Did you notice how, at the end of our routines, we added the line :

 (vla-delete (vla-item ssets "SS1"))

If we didn't delete the selection set, next time we ran the routine we would get a "*Selection Set Already Exists*" error. Now adding this line is fine if everything runs correctly in our routine, but what happens if the user Cancels or there is another error that causes our program not to reach this line.
Have a look at this :

```
(defun selset_test ()

(vl-load-com)

(setq acadDocument (vla-get-activedocument
     (vlax-get-acad-object)))

(setq ssets (vla-get-selectionsets acadDocument))

(setq flag nil)

(vlax-for item ssets

(if (= (vla-get-name item) "newsset")
        (setq flag T)
);if

);
```

```
(if flag

        (vla-delete (vla-item ssets "newsset"))

);if

    (setq newsset (vla-add ssets "newsset"))

);defun
```

This routine loops through each selection set in the selection set collection.
If it finds the selection set (*newset*), it deletes it and then creates it. If it does not find it, it simple creates it.

A better way of achieving the same result, is to make use of the (*vl-Catch-All-Apply*) function :

```
(defun selset_test1 ()

(vl-load-com)

(setq acadDocument (vla-get-activedocument
      (vlax-get-acad-object)))

(setq ssets (vla-get-selectionsets acadDocument))

(if (vl-catch-all-error-p (vl-catch-all-apply 'vla-item (list ssets "$Set")))

    (setq newSet (vla-add ssets "$Set"))

        (progn

            (vla-delete (vla-item ssets "$Set"))

            (setq newSet (vla-add ssets "$Set"))

        );progn

);if
```

Have a wee look at "Error Trapping" for a more detailed explanation of the
(*vl-Catch-All-Apply*) function.

 Well, that's about it with "Selecting Objects." Not too bad, hey?
(I can hear you groaning from here in Africa, Hee, hee, hee.)

Collections

All Objects in AutoCAD are grouped into collections. If you have a look at the AutoCAD Object Model, you will find all the Layers within the Layers collection, all the Blocks within the Blocks collection, etc. etc.

This tutorial will show you how to first, create a reference to the required collection, secondly, extract objects stored within the collection, and last but not least, manipulate these Objects.

Let's start right at the bottom of the Object Model with the Documents collection. Open any two drawings in AutoCAD then open the Visual List Editor and enter this :

_$ (vl-load-com)

_$ (setq acadObject (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00adc088>

Let's have a look at the "*acadObject*" in the Inspect window :

The object that we are interested in at this stage is the Documents object. Let's drill down to it :

_$ (setq acadDocuments (vla-get-documents acadObject))
#<VLA-OBJECT IAcadDocuments 01f585d0>

Have look at the variable "*acadDocuments*" in the inspect window :
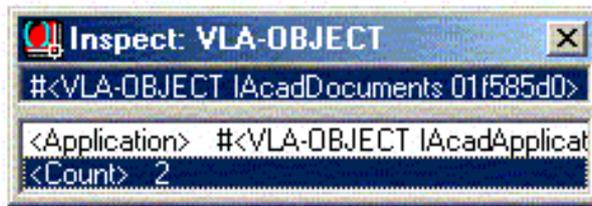
As you can see, this collection contains 2 objects. But how do we access these objects? Copy and paste the following coding and save it as "*tempList .lsp*" :

```
 (defun tempList (theObject / item dwgName)

   (vl-load-com)

   (setq theList '())

   (vlax-for item theObject

     (setq dwgName (vlax-get-property item 'Name))

     (setq theList (append (list dwgName) theList))

   );

   (setq theList (reverse theList))

(princ)

);defun
```

Don't worry to much at this stage about how this function works. We'll get to that later. Load the function and then type the following :

_$ (tempList acadDocuments)

Now examine the variable "theList" :

_$ theList
("Kenny.dwg" "is handsome.dwg")

This list now holds the contents of the Documents collection, or in other words, all the drawings you have open within AutoCAD..

**Let's go further in the Object Model and have a look at the Layers collection.
We first get a reference to the "Active Document" :**

_$ (vl-load-com)
_$ (setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

Next, a reference to the Layer's collection :

(setq theLayers (vla-get-layers acadDocument))
#<VLA-OBJECT IAcadLayers 01f5b0a4>

Now, let's extract all the Layer names into a list :

_$(tempList theLayers)
_$ theList
("0" "1" "2" "3" "4" "5" "6" "7" "DEFPOINTS" "8" "9" "10")

Great, we now have a list of all our Layers within the current drawing.

To manipulate objects within a collection, we first need to iterate through the collection and extract a reference to the object, or objects that we are interested in. Visual Lisp provides us with a few functions to help us with this task. Let's say for example that we wanted to change all the Layers in the drawing to Color 3. Consider this :

```
(defun laycol ()

(vl-load-com)

(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

(setq theLayers (vla-get-layers acadDocument))

(setq i 0)

   (repeat (vla-get-count theLayers)

       (setq aLayer (vla-item theLayers i))

       (vla-put-color alayer 3)

       (setq i (1+ i))

   );repeat

(princ)

);defun
```

Here we've used the standard AutoLisp function "*repeat*" to loop through the collection. We used the "*vla-get-count*" function to count the number of objects in the collection, and the function "*vla-item*" function to extract each object from the collection.

On the next page we'll have a look at a few more functions that will make your life a lot easier when dealing with collections.

On the first page we looked at changing all the Layers Color property using the "repeat" command. Here's a much easier and quicker way :

```
(defun laycol1 ()

(vl-load-com)

(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

(setq theLayers (vla-get-layers acadDocument))

(vlax-for item theLayers

        (vla-put-color item 3)

)

(princ)

);defun
```

"*vlax-for*" allows you to loop through each item in the collection, without having to count the number of objects and doing away with the need for loop control. This is the same method I used in the "*tempList*" function that we used earlier on in this tutorial.

Here's another example. Would you like to ensure that all your Layers are On before performing a certain function? Have a look at this :

```
(defun layeron ()

(vl-load-com)

(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

(setq theLayers (vla-get-layers acadDocument))

(vlax-for item theLayers

     (vlax-put-property item "LayerOn" ':vlax-true)

)

(princ)

);defun
```

One word of warning. *DO NOT* Add or Remove objects whilst iterating through a collection. This can and will cause errors.

One of the most powerful commands in our collections arsenal is the "*vlax-map-collection*" function. If you are not familiar with the "*mapcar*" and "*lambda*" functions, I suggest you read my tutorial on these functions.

Load and run this in the Visual Lisp Editor :

```
(defun layer-dump ()

(vl-load-com)

(setq theList '())

(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

(setq theLayers (vla-get-layers acadDocument))

(vlax-map-collection theLayers 'vlax-dump-object)

(princ)

);defun
```

This will dump all properties of all the Layer objects in your drawing to the console screen.

You are not limited to Visual Lisp functions within a "*vlax-map-collection*" call. You can also use your own user defined function. Let's say for some reason we wanted to make a list of all Layers in your drawing, switch all Layers On and change every Layer in the drawing to Color "*5*". This is how you could do it :

```
(defun layerMap ()

(vl-load-com)

(setq theList '())

(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

(setq theLayers (vla-get-layers acadDocument))

(vlax-map-collection theLayers 'layer-mod)

(princ)

);defun
```

```
(defun layer-mod (theLayer)

  (setq dwgName (vlax-get-property theLayer 'Name))

  (setq theList (append (list dwgName) theList))

  (setq theList (reverse theList))

  (vlax-put-property theLayer "LayerOn" ':vlax-true)

  (vla-put-color thelayer 5)

);defun
```

 As you can see, each Layer object is passed to the function "*layer-mod*" as the argument. You could also write this as an inline function using "*lambda*" :

```
(defun layerMap1 ()

(vl-load-com)

(setq theList '())

(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

(setq theLayers (vla-get-layers acadDocument))

(vlax-map-collection theLayers

  '(lambda (theLayer)

    (setq dwgName (vlax-get-property theLayer 'Name))

    (setq theList (append (list dwgName) theList))

    (setq theList (reverse theList))

    (vlax-put-property theLayer "LayerOn" ':vlax-true)

    (vla-put-color thelayer 5)

  );lambda

);vlax-map-collection

(princ)

);defun
```

Want to add to a collection? Let's add a new Layer to our drawing :

```
(defun layerAdd ()

(vl-load-com)

(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

(setq theLayers (vla-get-layers acadDocument))

(setq newLayer (vla-add theLayers "Steel"))

(vla-put-color newLayer 5)

(vla-put-linetype newLayer "Dashed2")

(princ)

);defun
```

This routine creates a new Layer named "*Steel*" with Color "*5*" and a Linetype of "*Dashed2*". If the Layer already exists, this routine simply does nothing.

To delete a Layer from the collection, simply use the "delete" method :

_$ (vla-delete newLayer)

Be careful though, if any objects within your drawing are referencing this Layer, you will get an error :

_$ (vla-delete newLayer)
; error: Automation Error. Object is referenced by other object(s)

Right, I've had enough of talking about collections. Now I'm going to check out my beer collection. It's not very big as I only collect the full ones. The empty ones I throw away. Ta, ta for now.......

Reactors

 "What the heck is a reactor?
I asked myself exactly the same question when I first came across the phrase. "It must be another name for an Event", I said to myself. I was wrong, again!!!
A reactor is an Object you attach to AutoCAD Objects to have AutoCAD notify your application when a particular event, or events occur.
(Remember, that an AutoCAD Object can be an Object within your drawing, the drawing itself, or even AutoCAD, the application - just thought I'd tell you).
This is the general procedure :

- An Event occurs in your drawing.
- AutoCAD notifies the Reaction associated with that Event.
- The reaction then notifies your application, known as a callback function, passing to it certain information applicable to the Event in the form of arguments.
- Your application then does it's thing using the information passed to it from the Reactor.

So, in simple terms, a Reactor is the "link" between the Event and the callback function.

Before we get into looking at the different types of AutoCAD Reactors and Events, let's have a look at a fairly straightforward Reactor to give you an idea of how simple they can be. Before you start though, you must ask yourself two questions :

1. **Before, after or during what Event do you want your function to occur?**
2. **What do you want your function to do?**

For this example, I will answer both these questions on your behalf.

1. *"I want my function to run every time a drawing Save is completed".*
2. *"I want the function to tell me what the name of the drawing is, and how big the drawing is".*

O.K. Looking at the Reactor listings, (*Refer to Visual Lisp Reference*), under the Editor Reactor Types, I find a reactor called "*vl-dwg-reactor*". This reactor, I am led to believe, responds to the "Save Complete" event of a drawing, and returns callback data of a string containing the actual file name used for the save.
Hey, this is just what we are looking for. Let's order two to take-away.

Enough of this childish wit!!
Let's have a look at the reactor definition and syntax first :

vlr-dwg-reactor - Constructs an editor reactor object that notifies of a drawing event.

<div align="center">

(vlr-dwg-reactor *data callbacks*)

</div>

Arguments :

data : AutoLisp data to be associated with the reactor object, or *nil*, if no data

callbacks : A list of pairs of the following form :

(event-name . callback_function)

where event-name is one of the symbols listed in the "DWG reactor events" table, and callback_function is a symbol representing the function to be called when the event fires. Each callback function will accept two arguments :
reactor_object - the VLR object that called the callback function,
list - a list of extra data elements associated with the particular event.

I don't know about you? But, whew.........!!!
Right, let's try and put this into simple English. Let's look at the syntax again :

(vlr-dwg-reactor *data callbacks*)

The first part "vlr-dwg-reactor" is easy. This is the name of the reactor type. This name will be sent to your call back function.

The first argument "data", is User Application Data. We usually set this to a reactor name of our choosing. This way we can distinguish between reactors if an Objects has multiple reactors attached.

The second argument "callbacks" is a straightforward list of dotted pairs.

- The first element of the list is the name of the reactor "event" that will trigger the reactor and then call your callback function.
- The second element, is the name of your Callback function.

This is what our reactor function will look like :

(vlr-dwg-reactor "Save Complete" '((:vlr-savecomplete . savedrawinginfo)))

Or, graphically :

The Reactor Type

Application Data
(user Reactor Name)

Reactor Event

Callback Function

```
(vlr-dwg-reactor "Save Complete" '((:vlr-savecomplete . Savedrawinginfo)))
```

Calling Reactor Name
and
Event parameter information

Callback Function
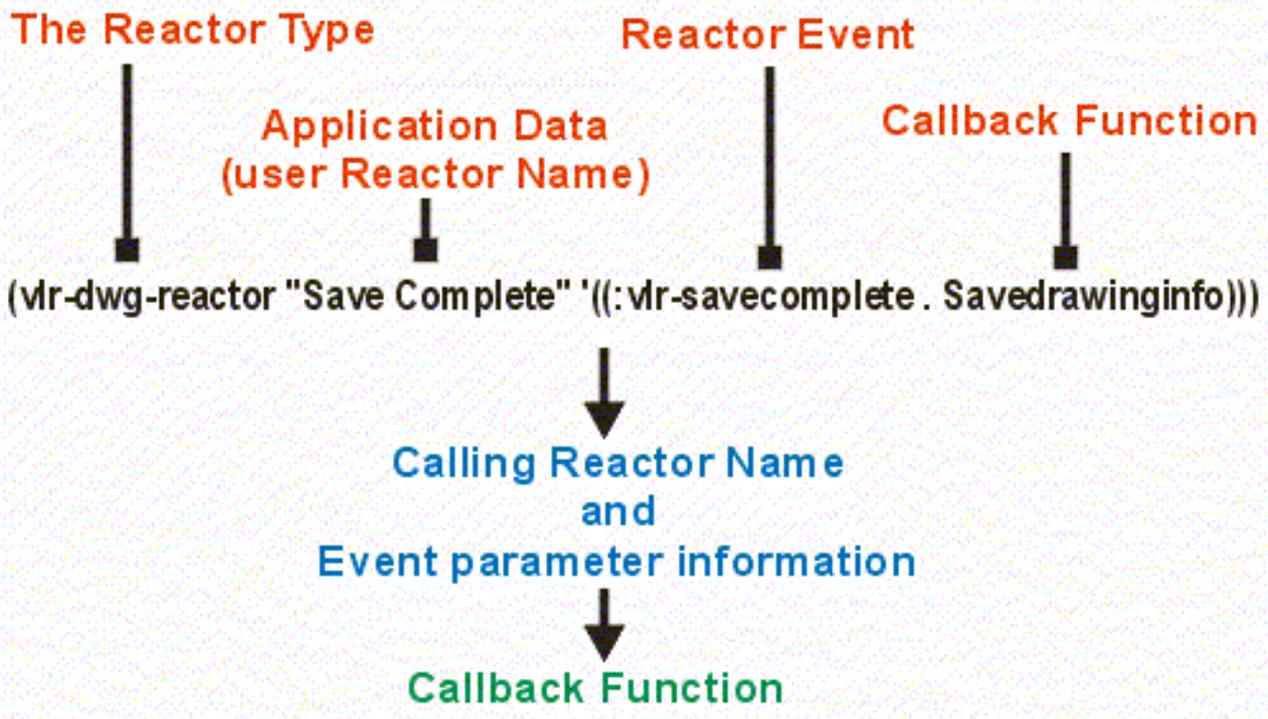
Let's have a look at our Reactor Function in action. Copy and Paste this coding into the Visual Lisp Editor and save it as "SaveDrawingInfo.Lsp".
Next Load the application, but DO NOT run it.

```
(vl-load-com)

;***********************************************************

;setup and intilise the reactor
(vlr-dwg-reactor "Save Complete" '((:vlr-savecomplete . savedrawinginfo)))

;***********************************************************

(defun saveDrawingInfo (calling-reactor commandInfo / dwgname filesize
                        reactType reactData reactCall
                        reactEvent reactCallback)

;get the reactor Object
(setq reactInfo calling-reactor

    ;get the reactor Type
    reactType (vl-symbol-name (vlr-type reactInfo))

    ;get the Application Data
    reactData (vlr-data reactInfo)

    ;get the Callback list
    reactCall (car (vlr-reactions reactInfo))

    ;extract the Event Reactor
    reactEvent (vl-symbol-name (car reactCall))
```

```
    ;extract the Callback Function
    reactCallback (vl-symbol-name (cdr reactCall))

);setq

;get the Drawing Name
(setq dwgname (cadr commandInfo)

    ;extract the filesize
    filesize (vl-file-size dwgname)

);setq

;display the Drawing Name and Size
(alert (strcat "The file size of " dwgname " is "

(itoa filesize) " bytes."))

;Display the Reactor Information
(alert

  (strcat
    "A " "\"" reactType "\"" " named " "\"" reactData "\"" "\n"
    "was triggered by a " "\"" reactEvent "\"" " event call." "\n"
    "Callback Data was passed to the" "\n"
    "\"" reactCallback "\"" " call back function."))

(princ)

);defun

;****************************************************

(princ)

;****************************************************
```

Once the application is loaded, return to AutoCAD and save the drawing.
This dialog will appear :

Followed by this dialog :
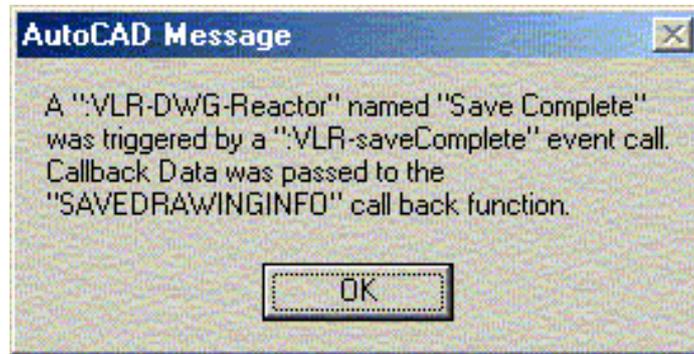
Do you notice the "calling-reactor" and "commandInfo" argument declarations? This information, the Reactor Object Name and the Event Parameter Information is passed to our Callback Function from the Reactor.
Clever hey?

On the next Page, we'll have a look at some Drawing Reactors.

Another Editor Type of reactor is the "VLR-Command-Reactor". This reactor notifies us of a Command Event. In this instance we will make use of the "vlr-commandEnded" reactor event which returns a event parameter list containing a string identifying the command that has been cancelled.

Every time a drawing is plotted, our call back function will first, save the drawing, and secondly save the drawing to a backup directory, namely C:/Backup. Let's have a look at the coding :

```
(prompt " \nLoad Only....Do NOT Run...")

(vl-load-com)

;****************************************************
(vlr-command-reactor

        "Backup After Plot" '((:vlr-commandEnded . endPlot)))

;****************************************************
(defun endPlot (calling-reactor endcommandInfo /

                  thecommandend drgName newname)

(setq thecommandend (nth 0 endcommandInfo))

        (if (= thecommandend "PLOT")

        (progn

        (setq acadDocument (vla-get-activedocument
                            (vlax-get-acad-object)))

        (setq drgName (vla-get-name acadDocument))

        (setq newname (strcat "c:\\backup\\" drgName))

        (vla-save acadDocument)

        (vla-saveas acadDocument newname)

        );progn

        );if

(princ)

);defun

;****************************************************

(princ)
```

A word of warning!!! Did you notice how I used ActiveX statements and functions to "*Save*" and "*SaveAs*". You cannot use interactive functions from within a reactor 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*", "*ensel*", and "*getkword*", as well as "*selection set*" operations and the "*command*" function.

Here's another interesting command event reactor :

When a user adds Text or Hatch to the drawing, the layer will automatically change to Layer "4" or Layer "6" respectively. When the command is completed, or cancelled, the user is returned to the original Layer he was on before he started the command. Save the file as "LayMan.lsp", BUT please remember, that as this routine contains reactors, you must only Load it and NOT Run it. If you want to use this routine on a permanent basis, you'll have to ensure that it is loaded at startup. There is also no checking to ensure that the layers exist, are frozen, switched off, etc. and no other form of error checking. I've got to leave something for you to do!!!

```
(prompt " \nLoad Only....Do NOT Run...")
(vl-load-com)


;**************************************
(vlr-command-reactor
        nil '((:vlr-commandWillStart . startCommand)))
(vlr-command-reactor
        nil '((:vlr-commandEnded . endCommand)))
(vlr-command-reactor
        nil '((:vlr-commandCancelled . cancelCommand)))
;**************************************************
(defun startCommand (calling-reactor startcommandInfo /
                        thecommandstart)
(setq OldLayer (getvar "CLAYER"))
(setq thecommandstart (nth 0 startcommandInfo))
(cond
  ((= thecommandstart "TEXT") (setvar "CLAYER" "4"))
  ((= thecommandstart "MTEXT") (setvar "CLAYER" "4"))
  ((= thecommandstart "DTEXT") (setvar "CLAYER" "4"))

 ((= thecommandstart "HATCH") (setvar "CLAYER" "6"))
 ((= thecommandstart "BHATCH") (setvar "CLAYER" "6"))
);cond
(princ)
);defun
;*************************************************

(defun endCommand (calling-reactor endcommandInfo /
                        thecommandend)
(setq thecommandend (nth 0 endcommandInfo))
(cond
```

```
    ((= thecommandend "TEXT") (setvar "CLAYER" OldLayer))
    ((= thecommandend "MTEXT") (setvar "CLAYER" OldLayer))
    ((= thecommandend "DTEXT") (setvar "CLAYER" OldLayer))
    ((= thecommandend "HATCH") (setvar "CLAYER" OldLayer))
    ((= thecommandend "BHATCH") (setvar "CLAYER" OldLayer))

);cond
 (princ)
);defun
;****************************************************
(defun cancelCommand (calling-reactor cancelcommandInfo /
                      thecommandcancel)
(setq thecommandcancel (nth 0 cancelcommandInfo))
(cond
   ((= thecommandcancel "TEXT") (setvar "CLAYER" OldLayer))
   ((= thecommandcancel "MTEXT") (setvar "CLAYER" OldLayer))
   ((= thecommandcancel "DTEXT") (setvar "CLAYER" OldLayer))
   ((= thecommandcancel "HATCH") (setvar "CLAYER" OldLayer))
   ((= thecommandcancel "BHATCH") (setvar "CLAYER" OldLayer))
);cond
(princ)
);defun
;****************************************************
(princ)
```

Did you notice that this application used three command reactors with three different command events. We could have incorporated all three reactor events and call back functions under one command reactor type, but I prefer to leave them separate for clarity and ease of debugging.

O.K. that's enough of command reactors. Let's have a look at Object Reactors.

Object Reactors, or "VLR-Object-Reactor", fall under general reactor types. They are almost identical in functionality to Drawing and Command reactors except for a couple of things! They need to include a reference to the Object that will be reacted upon, (Crikey, that sounds terrible!!) and the reference to the Object needs to be created before the reactor is called. Let's have a look at the syntax of an Object reactor :

(vlr-object-reactor *owners data callback*)

The "*data*" and "*callback*" arguments, we are familiar with. But what is the "*owner*" argument? This is an AutoLisp list of Visual Lisp Objects identifying the drawing Objects to be watched. In other words, a reference to the Object that contains the reactor.
The reactor event we are going to use is the "*:vlr-modified event*", and our Callback function will be named "*print-length*".
Have a look at the coding for our reactor :

(vlr-object-reactor (list myLine) "Line Reactor" '((:vlr-modified . print-length)))

As I mentioned earlier though, we need to have a reference to the Object before we can call this statement. Consider the following :

```
(vl-load-com)

;*************

(defun line-draw ()

(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))

(setq mspace (vla-get-modelspace acadDocument))

(setq apt (getpoint "Specify First Point: "))

(setq pt (getpoint "Specify next point: " apt))

(setq myLine (vla-addline mspace (vlax-3d-point apt)(vlax-3d-point pt)))

(setq lineReactor (vlr-object-reactor (list myLine)
"Line Reactor" '((:vlr-modified . print-length))))

(princ)

);defun
```

We started off by drawing a line. As the line was created from scratch, and created using Visual Lisp functions, we already have a reference to the line Object. (*myLine*). We can now safely run our reactor function and attach it to our Line.
"But where is the Callback function?"
Hah, I was waiting for that. We've made the Callback function a separate function for one

**main reason. If we didn't, every time we ran the application it would prompt us to draw a
new line. So, what we have to do now, is link the reactor function to our Callback function
so that when our line is modified, only the Callback function is put into motion. The
reactor sends three arguments to the Callback function, the notifier-object (*our line*), the
reactor-object (*:vlr-modified*), and the event parameter-list which in this case is *nil*.**

Here's the coding for the Callback function :

```
(defun print-length (notifier-object reactor-object parameter-list)

(cond
        ((vlax-property-available-p notifier-object "Length")
         (alert (strcat "The length is now "
                        (rtos (vla-get-length notifier-object)))))
);cond

(princ)

);defun

(princ)
```

**Copy all of this coding into one file and save it as "*Line-Draw.Lsp*". Now load "*Line-
Draw.Lsp*" and then run (*line-draw*). Draw a single line when prompted. Now stretch the
line so that it's length changes. A dialog will appear displaying the new length of the line :**
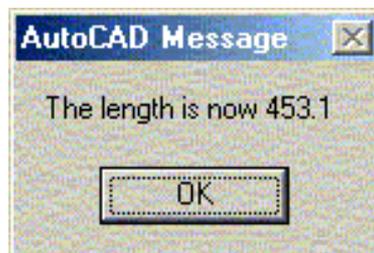
In essence, this is what happened :

- We loaded "*Line-draw.Lsp*" and all functions contained within were placed into memory.
- We ran (*line-draw*) which prompted us to draw a line. The reactor was then loaded and linked to both the line Object and the Callback function.
- As the Callback function "*print-length*" was also loaded into memory, every time we modify the line Object, the Callback function is processed and the length of the line is displayed.

**Did you notice how we checked that our Object had a *"Length"* Property before
continuing? Good idea, as this validation can save lot's of problems.**

"*But what happens when I close my drawing? Will I lose all my reactors?*"

Good questions. Reactors can be transient or persistent. Transient reactors are lost when the drawing closes and this is the default reactor mode. Persistent reactors are saved with the drawing and exist when the drawing is next open.

You can use the "*vlr-pers*" function to make a reaction persistent. To remove a persistence from a reactor and make it transient, use the "*vlr-pers-release*" function. To determine whether a reactor is persistent or transient, use the "*vlr-pers-p*" function. Each function takes the reactor Object as it's only argument :

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

If successful "*vlr-pers*" returns the specified reactor Object.

Note : A reactor is only a link between an event and a Callback function. The Callback function 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.
In other words, if we made our reactor "*lineReactor*" persistent, we would have to ensure that the Callback function "*print-length*" was loaded every time the drawing containing our lines with reactors was opened.

Visual Lisp and Menu's.

This tutorial was very kindly written by Stig Madsen. and is published, here on AfraLisp, with permission.

Before starting on this tutorial, I presume that you are familiar with AutoCAD menu's and that you have a bit of experience in modifying menu's or creating your own partial menu's. It will also make things a lot easier for you if you have a basic understanding of AutoCAD's macro language.
I
O.K. enough chirping from me, over to Stig....

This code will do the following:

Create an empty menu file, vbamenu.mns, if it doesn't already exist

Create a new menugroup called "VbaMenu" if it doesn't already exist

Create a new pulldown menu in the menugroup "VbaMenu"

Populate the pulldown menu with items of our own choice

Load and install the menugroup into AutoCAD

Save the menufile as "VbaMenu.mns" and compile it to "VbaMenu.mnc"
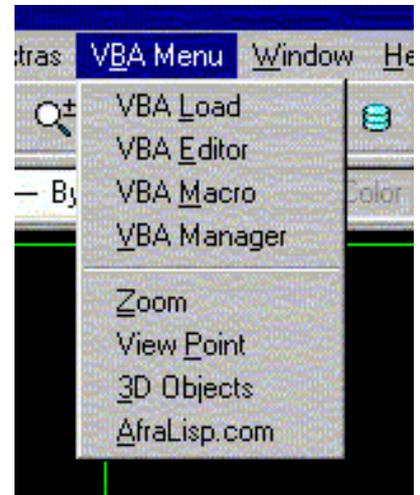
 Following code has a nested *DEFUN* in it. This technique is used to avoid passing around variables to different routines. Everything in the nested subroutine *CreateMenu* gets executed if the pulldown menu we are looking for doesn't exist. Of course, it could be placed within a lengthy *PROGN* statement at the end of the test, but this way the code does a much more obvious branching.

Notice that our new menufile isn't actually loaded before we make sure that it isn't already loaded. Because AutoCAD deals with standard Windows issues when setting up menus, strange things can happen if we force a multiple load of the same menu.

```
(defun C:VBATOOLBARMENU (/ fn acadobj thisdoc menus flag currMenuGroup
newMenu



              newMenuItem openMacro


          )


  ;; CreateMenu is a nested DEFUN that is executed if our "VbaMenu"
```

```
;; pulldown menu doesn't exist. A test for the presence of this

;; pulldown menu is done in the main code

(defun createMenu ()

  ;; Add a new popUpMenu to currMenuGroup, i.e. to "VbaMenu"

  (setq newMenu (vla-add (vla-get-menus currMenuGroup) "V&BA Menu"))

  ;;-------------------------------------------------------------

  ;; create the first pulldown item, vbaload

  (setq openMacro (strcat (chr 3) (chr 3) (chr 95) "vbaload" (chr 32)))

  (setq newMenuItem

        (vla-addMenuItem

          newMenu

          (1+ (vla-get-count newMenu))

          "VBA &Load"

          openMacro

        )

  )

  (vla-put-helpString newMenuItem "Load a VBA Application")

  ;;-------------------------------------------------------------

  ;; create the second pulldown item, vbaide

  (setq openMacro (strcat (chr 3) (chr 3) (chr 95) "vbaide" (chr 32)))

  (setq newMenuItem

        (vla-addMenuItem

          newMenu
```

```
                (1+ (vla-get-count newMenu))

                "VBA &Editor"

                openMacro

            )

)

(vla-put-helpString newMenuItem "Switch to the VBA Editor")

;;-------------------------------------------------------

;; create the third pulldown item, vbarun

(setq openMacro (strcat (chr 3) (chr 3) (chr 95) "vbarun" (chr 32)))

(setq newMenuItem

        (vla-addMenuItem

          newMenu

          (1+ (vla-get-count newMenu))

          "VBA &Macro"

          openMacro

        )

)

(vla-put-helpString newMenuItem "Run a VBA Macro")

;;-------------------------------------------------------

;; create the fourth pulldown item, vbaman

(setq openMacro (strcat (chr 3) (chr 3) (chr 95) "vbaman" (chr 32)))

(setq newMenuItem

        (vla-addMenuItem
```

```
            newMenu

            (1+ (vla-get-count newMenu))

            "&VBA Manager"

            openMacro

        )

    )

    (vla-put-helpString newMenuItem "Display the VBA Manager")

    ;;-----------------------------------------------------------

    ;; insert a separator after the fourth menu item

    (vla-AddSeparator newMenu 5)

    ;;-----------------------------------------------------------

    ;; create a simple menu macro

    (setq

      openMacro (strcat (chr 3) (chr 3) (chr 95) "zoom" (chr 32) "w" (chr
32))

    )

    (setq newMenuItem

          (vla-addMenuItem

            newMenu

            (1+ (vla-get-count newMenu))

            "&Zoom"

            openMacro

        )

    )
```

```
(vla-put-helpString newMenuItem "Zoom Window")

;;----------------------------------------------------------------

;; create a menu item that loads and runs an AutoLISP routine

(setq openMacro (strcat (chr 3)

                        (chr 3)

                        (chr 95)

                        "(if (not c:ddvpoint) (load \"ddvpoint\")"

                        (chr 32)

                        "ddvpoint"

                )
)

(setq newMenuItem

        (vla-addMenuItem

          newMenu

          (1+ (vla-get-count newMenu))

          "View &Point"

          openMacro

        )

)

(vla-put-helpString newMenuItem "View Point")

;;----------------------------------------------------------------

;; create a menu item that calls an Image menu

(setq openMacro (strcat (chr 3)
```

```
                                                (chr 3)

                                                (chr 95)

                                                "$I=image_3dobjects $I=*"

                            )

)

(setq newMenuItem

            (vla-addMenuItem

                newMenu

                (1+ (vla-get-count newMenu))

                "&3D Objects"

                openMacro

            )

)

(vla-put-helpString newMenuItem "3D objects")

;;------------------------------------------------------------

;; create a menu item with a hyperlink

(setq openMacro (strcat (chr 3)

                                    (chr 3)

                                    (chr 95)

                                    "browser"

                                    (chr 32)

                                    "www.afralisp.com"

                                    (chr 32)

                            )
```

```
  )

  (setq newMenuItem

        (vla-addMenuItem

          newMenu

          (1+ (vla-get-count newMenu))

          "&AfraLisp.com"

          openMacro

        )

  )

  (vla-put-helpString newMenuItem "Go visit this awesome place, or
else!")

  ;;-------------------------------------------------------------

  ;; insert the pulldown menu into the menu bar, third from the end

  (vla-insertInMenuBar

    newMenu

    (- (vla-get-count (vla-get-menuBar acadobj)) 2)

  )

  ;; re-compile the VBAMENU menu - VBAMENU.MNC

  (vla-save currMenuGroup acMenuFileCompiled)

  ;; save it as a MNS file

  (vla-save currMenuGroup acMenuFileSource)

)
```

```lisp
;; First, check to see if our menu file "VbaMenu.mns" already

;; exists. If it doesn't then simply make an empty file that

;; we can later write our menu definition to

(setq flag nil)

(if (not (findfile "VbaMenu.mns"))

  (progn

    (setq fn (open "VbaMenu.mns" "w"))

    (close fn)

  )

)

;; Get hold of the application object - we will use it to

;; retrieve the menuGroups collection, which is a child object

;; of the application

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

;; Get the active document - also a child of the application

(setq thisdoc (vla-get-activeDocument acadobj))

;; Get all menugroups loaded into AutoCAD

(setq menus (vla-get-menuGroups acadobj))

;; Now we could use VLA-ITEM to test if "VbaMenu" exists among

;; all loaded menugroups with (vla-item menus "VbaMenu").

;; Instead, as a friendly service, we want all loaded menus to

;; be printed to the screen and at the same time we might as well

;; use it to set a flag if "VbaMenu" is among the loaded menus

(princ "\nLoaded menus: ")
```

```
(vlax-for n menus

  (if (= (vla-get-name n) "VbaMenu")

    (setq flag T)

  )

  (terpri)

  (princ (vla-get-name n))

)

;; If VbaMenu wasn't among the loaded menus then load it

(if (null flag)

  (vla-load menus "VbaMenu.mns")

)

(setq currMenuGroup (vla-item menus "VbaMenu"))

;; If no popUpMenus exist in VbaMenu then go create one -

;; otherwise exit with grace. In this example we merely check

;; if the number of popup menus in "VbaMenu" is greater than 0.

;; A safer way to test for its presence would be to set up a

;; test for its name, "V&BA Menu":

;; (vla-item (vla-get-menus currMenuGroup) "V&BA Menu")

(if (<= (vla-get-count (vla-get-menus currMenuGroup)) 0)

  (createMenu)

  (princ "\nThe menu is already loaded")

)

(princ)
```

)

Now navigate your way to VbaMenu.mns and open it. You should see something like this :

//
// AutoCAD menu file - D:\drawings\VbaMenu.mns
//

***MENUGROUP=VbaMenu

***POP2
ID_mnuVBA Menu [V&BA Menu]
ID_VBA Load [VBA &Load]^C^C_vbaload
ID_VBA Editor [VBA &Editor]^C^C_vbaide
ID_VBA Macro [VBA &Macro]^C^C_vbarun
ID_VBA Manager [&VBA Manager]^C^C_vbaman
[--]
ID_Zoom [&Zoom]^C^C_zoom w
ID_View Point [View &Point]^C^C_(if (not c:ddvpoint) (load "ddvpoint") ddvpoint
ID_3D Objects [&3D Objects]^C^C_$I=image_3dobjects $I=*
ID_AfraLisp.com [&AfraLisp.com]^C^C_browser www.afralisp.com

***TOOLBARS

***HELPSTRINGS
ID_VIEW POINT [View Point]
ID_AFRALISP.COM [Go visit this awesome place, or else!]
ID_VBA MANAGER [Display the VBA Manager]
ID_VBA LOAD [Load a VBA Application]
ID_ZOOM [Zoom Window]
ID_VBA MACRO [Run a VBA Macro]
ID_3D OBJECTS [3D objects]
ID_VBA EDITOR [Switch to the VBA Editor]

//
// End of AutoCAD menu file - D:\drawings\Vbamenu.mns
//

You should also find VbaMenu.mnc and VbaMenu.mnr in the same folder. Even though the VbaMenu.mns didn't exist when we started, Visual Lisp has created it as well as all the coding necessary and compiled the other menu support files required for the menu to run.

Next we'll have a look at creating Toolbar menu's.

Using Stigs coding as a template, I've written the follow routine that creates a Toolbar menu. Please ensure that the toolbar bitmap files are within your AutoCAD support path before loading and running this application.

```lisp
(defun C:VBATOOLBARMENU (/ fn acadobj thisdoc menus flag currMenuGroup
newToolbar newToolbarButton openMacro
SmallBitmapName LargeBitmapName)

(vl-load-com)

;;; CreateToolbar is called if the Toolbar in question doesn't exist
  (defun createToolbar ()
    (setq newToolbar (vla-add (vla-get-toolbars currMenuGroup) "VBA Menu"))
    ;;-----------------------------------------------------------------
    ;; create the first Toolbar Button, VbaLoad
    (setq openMacro (strcat (chr 3) (chr 3) (chr 95) "vbaload" (chr 32)))
    (setq newToolbarButton (vla-addToolbarButton
            newToolbar
            (1+ (vla-get-count newToolbar))
            "VBA Load" "VBA Load" openMacro
          )
    )
    (setq SmallBitmapName "VbaLoad.bmp")
    (setq LargeBitmapName "VbaLoad.bmp")
    (vla-setBitmaps newToolbarButton SmallBitmapName LargeBitmapName)

    (vla-put-helpString newToolbarButton "Load a VBA Application")
    ;;-----------------------------------------------------------------
    ;; create the second Toolbar Button, Vbaide
    (setq openMacro (strcat (chr 3) (chr 3) (chr 95) "vbaide" (chr 32)))
    (setq newToolbarButton (vla-addToolbarButton
            newToolbar
            (1+ (vla-get-count newToolbar))
            "VBA Editor" "VBA Editor" openMacro
          )
    )
    (setq SmallBitmapName "Vbaide.bmp")
    (setq LargeBitmapName "Vbaide.bmp")
    (vla-setBitmaps newToolbarButton SmallBitmapName LargeBitmapName)

    (vla-put-helpString newToolbarButton "Switch to the VBA Editor")
    ;;-----------------------------------------------------------------
    ;; create the third Toolbar Button, Vbarun
    (setq openMacro (strcat (chr 3) (chr 3) (chr 95) "vbarun" (chr 32)))
    (setq newToolbarButton (vla-addToolbarButton
            newToolbar
            (1+ (vla-get-count newToolbar))
            "VBA Macro" "VBA Macro" openMacro
          )
    )
    (setq SmallBitmapName "Vbamacro.bmp")
    (setq LargeBitmapName "Vbamacro.bmp")
```

```
        (vla-setBitmaps newToolbarButton SmallBitmapName LargeBitmapName)

        (vla-put-helpString newToolbarButton "Run a VBA Macro")
        ;;-------------------------------------------------------------
        ;; create the fourth Toolbar Button, Vbaman
        (setq openMacro (strcat (chr 3) (chr 3) (chr 95) "vbaman" (chr 32)))
        (setq newToolbarButton (vla-addToolbarButton
                newToolbar
                (1+ (vla-get-count newToolbar))
                "VBA Manager" "VBA Manager" openMacro
            )
        )
        (setq SmallBitmapName "Vbaman.bmp")
        (setq LargeBitmapName "Vbaman.bmp")
        (vla-setBitmaps newToolbarButton SmallBitmapName LargeBitmapName)

        (vla-put-helpString newToolbarButton "Display the VBA Manager")
        ;;-------------------------------------------------------------

        ;; re-compile the VBATOOLBARMENU menu - VBATOOLBARMENU.MNC
        (vla-save currMenuGroup acMenuFileCompiled)
        ;; save it as a MNS file
        (vla-save currMenuGroup acMenuFileSource)
)

(setq flag nil)
(if (not (findfile "VbaToolbarMenu.mns"))
    (progn
        (setq fn (open "VbaToolbarMenu.mns" "w"))
        (close fn)
    )
)
;; get hold of the application object
;; we'll use it to reference the menuGroups collection
(setq acadobj (vlax-get-acad-object))
;; .. and get the active document
(setq thisdoc (vla-get-activeDocument acadobj))
;; get all menu groups loaded into AutoCAD
(setq menus (vla-get-menuGroups acadobj))
(princ "\nLoaded menus: ")
(vlax-for n menus
    (if (= (vla-get-name n) "VbaToolbarMenu")
        (setq flag T)
    )
    (terpri)
    (princ (vla-get-name n))
)
;; if VbaToolbarMenu wasn't among the loaded menus then load it
(if (null flag)
    (vla-load menus "VbaToolbarMenu.mns")
)
(setq currMenuGroup (vla-item menus "VbaToolbarMenu"))
```

```
  ;; if no Toolbars exist in VbaToolbarMenu then go create one
  ;; otherwise exit with grace
  (if (<= (vla-get-count (vla-get-menus currMenuGroup)) 0)
    (createToolbar)
    (princ "\nThe Vba Toolbar Menu is already loaded")
  )
  (princ)
)

(princ)
```

Your "VbaToolbarMenu.mns", should look like this :

//
// AutoCAD menu file - D:\drawings\VbaToolbarMenu.mns
//

***MENUGROUP=VbaToolbarMenu

***TOOLBARS
**VBA_MENU
ID_VBA_Menu_0 [_Toolbar("VBA Menu", _Floating, _Show, 168, 152, 1)]
ID_VBA_Load_0 [_Button("VBA Load", "VbaLoad.bmp", "VbaLoad.bmp")]^C^C_vbaload
ID_VBA_Editor_0 [_Button("VBA Editor", "Vbaide.bmp", "Vbaide.bmp")]^C^C_vbaide
ID_VBA_Macro_0 [_Button("VBA Macro", "Vbamacro.bmp", "Vbamacro.bmp")]^C^C_vbarun
ID_VBA_Manager_0 [_Button("VBA Manager", "Vbaman.bmp", "Vbaman.bmp")]^C^C_vbaman

***HELPSTRINGS
ID_VBA_MANAGER_0 [Disply the VBA Manager]
ID_VBA_LOAD_0 [Load a VBA Application]
ID_VBA_MACRO_0 [Run a VBA Macro]
ID_VBA_EDITOR_0 [Switch to the VBA Editor]

//
// End of AutoCAD menu file - D:\drawings\VbaToolbarMenu.mns
//

Your Toolbar should look like this :

Visual Lisp and Errors

In standard AutoLisp, if your program encounters an error of any sort, it passes to the *error* function only one thing, a description of the error, and your program then ends.
Let's have a quick look at this in action. Load and run this little routine :

```
(defun error-test1 ()

(setq int (getint "\nEnter Number : "))

(setq result (apply 'sqrt (list int)))

(alert (strcat "Result = " (rtos result)))

(princ)

);defun
```

Enter "4" at the command prompt. This should result in an alert dialog displaying the answer "*Result = 2*".
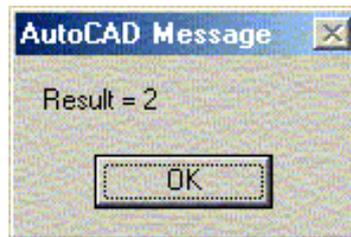
Now, run it again and enter "- 4".
Your program will come up with an error :

_$ (error-test1)
; error: function undefined for argument: -4

The reason being of course, is that you cannot determine the square root of a negative number.
Now let's add our error trap :

```
(defun error-test1 ()

(setq temperr *error*)

(setq *error* trap)

(setq int (getint "\nEnter Number : "))

(setq result (apply 'sqrt (list int)))

(alert (strcat "Result = " (rtos result)))

(princ)

);defun
```

```
(defun trap (errmsg)

        (setq *error* temperr)

        (alert errmsg)

        (alert "There was an error!!")

    (princ)

);defun
```

Load and run the program again. The error will be encountered and control will pass to the error trap which will display two alert dialogs, one displaying the error message, and one displaying a user defined error message :
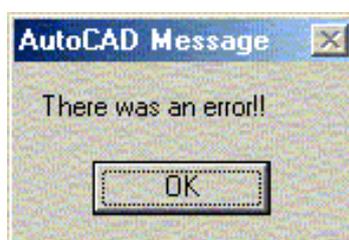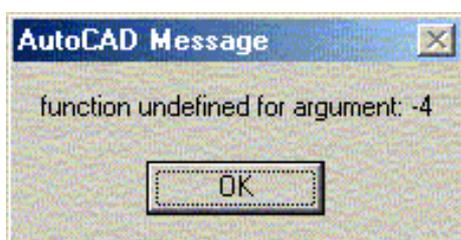
Your program will now stop. In a properly designed program, the error trap function would reset the system to the state it was in before your program started. (eg. snaps would be reset, system variables would be reset, etc).

Now this fine if you're using standard AutoLisp, but if you are using Visual Lisp function, this is another matter. Many of the Visual Lisp functions are designed to be used in the "*programming by exception*" style. This means they either return useful values if they succeed, or raise an exception if they fail (instead of returning an error value). If your program uses Visual Lisp functions, you must prepare it to catch exceptions, otherwise the program halts, leaving the user at the command prompt.
The advantage of this though, is that your program can intercept and attempt to process errors instead of allowing control to pass to the *error* function.
For this we would use the "*vl-catch-all-apply*" function which is designed to invoke any function, return the value from the function, and trap any error that may occur. You could call it an "*inline local error function.*"
The function requires two arguments :

■ a symbol identifying a function or "lambda" expression
■ a list or arguments to be passed to the calling function

Here's how we would use it in our program :

(setq result (vl-catch-all-apply 'sqrt (list int)))

Almost exactly the same as the "*apply*" function hey!

Now load and run the following, entering "- 4" at the command prompt to force and error :

```
(defun error-test2 ()

  (setq int (getint "\nEnter Number : "))

  (setq result (vl-catch-all-apply 'sqrt (list int)))

  (alert (strcat "Result = " (rtos result)))

  (princ)

);defun
```

An error message should appear at the Console prompt :

_$ (error-test2)
; error: bad argument type: numberp: #<%catch-all-apply-error%>

Have a look at the variable "*result*" in the Watch window :
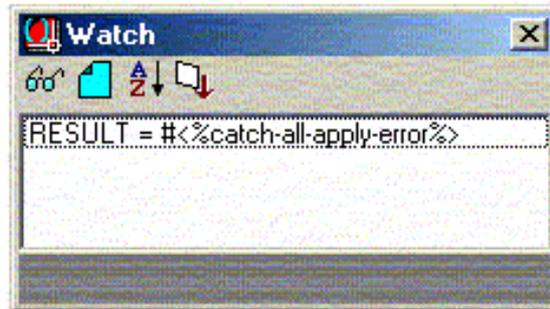
The variable contains an "*error object*"!!
If the program runs correctly, "*vl-catch-all-apply*" stores the return value in "*result*". If the call is unsuccessful, "*vl-catch-all-apply*" stores an error object in "*result*".
Using the "*vl-catch-all-error-p*" function, we can test for this "*error object*" :

```
(defun error-test2 ()

  (setq int (getint "\nEnter Number : "))

  (setq result (vl-catch-all-apply 'sqrt (list int)))

  (if (vl-catch-all-error-p result)

    (progn

      (setq int (abs int))

      (setq result (vl-catch-all-apply 'sqrt (list int)))

    );progn

  );if

  (alert (strcat "Result = " (rtos result)))
```

```
    (princ)

);defun
```

Load and run this routine entering "*- 4*" at the command prompt. This should have caused an error, but because we used the "*vl-catch-all-apply*" function, the error was trapped and by testing for the error using "*vl-catch-all-error-p*", we were able to rectify the problem by using the "abs" function.
Let's add our normal error function :

```
(defun error-test2 ()

   (setq temperr *error*)

   (setq *error* trap)

   (setq int (getint "\nEnter Number : "))

   (setq result (vl-catch-all-apply 'sqrt (list int)))

   (if (vl-catch-all-error-p result)

     (progn

       (setq int (abs int))

       (setq result (vl-catch-all-apply 'sqrt (list int)))

     );progn

   );if

   (alert (strcat "Result = " (rtos result)))

   (princ)

);defun

(defun trap (errmsg)

        (setq *error* temperr)

        (alert errmsg)

        (alert "There was an error!!")

    (princ)

);defun
```

Load and run the program, again entering "*- 4*" at the command prompt.
No error, everything runs smoothly. Now run the program again, but this time hitting "Esc" at the command prompt. This error is not an ActiveX error, therefore the error is passed to our

conventional error trap to be processed.

Hey, we've got two "*error*" functions to play with now!

Another good example of when to us the "*vl-catch-all-apply*" function, is when dealing with "*Selection Sets*" using Visual Lisp.

When you first create a selection set in Visual Lisp, all is well and good. But, if you try to create a selection set that already exists, you will raise an error. Here's a way around that problem :

```
(defun selset_test1 ()

(vl-load-com)

(setq acadDocument (vla-get-activedocument
     (vlax-get-acad-object)))

(setq ssets (vla-get-selectionsets acadDocument))

(if (vl-catch-all-error-p (vl-catch-all-apply 'vla-item (list ssets "$Set")))

  (setq newSet (vla-add ssets "$Set"))

    (progn

       (vla-delete (vla-item ssets "$Set"))

       (setq newSet (vla-add ssets "$Set"))

    );progn

);if

);defun
```

If the selection sets does not exists, the selection set is created. If it does exist, it is first deleted and then created.

Visual Lisp and Layers

I have had so many queries lately regarding Visual Lisp and Layer's, that I decided to dedicated a complete section dedicated to this subject. Please remember though, that what you read here is not all encompassing in regards to Layers and could possible be added to over time. (in other words, I'm making excuses in case I miss anything).

Layer's in AutoCAD are contained within the Layer's Collection which is stored in the Document Object which is part of the Documents Collection which is part of the Application Object or AutoCAD itself. Confused? Have a look at an extract from the AutoCAD Object Model :

```
Application(Object)
        |
        |
        |
        |------- Documents[Collection]
        |             |
        |             |
        |             |
        |        Document(Object)
        |             |
        |             |
        |             |------ Layers-------Layer
        |             |      [Collection]    (Object)
        |             |
        |             |
        |             |
        |             |
        |             |
```
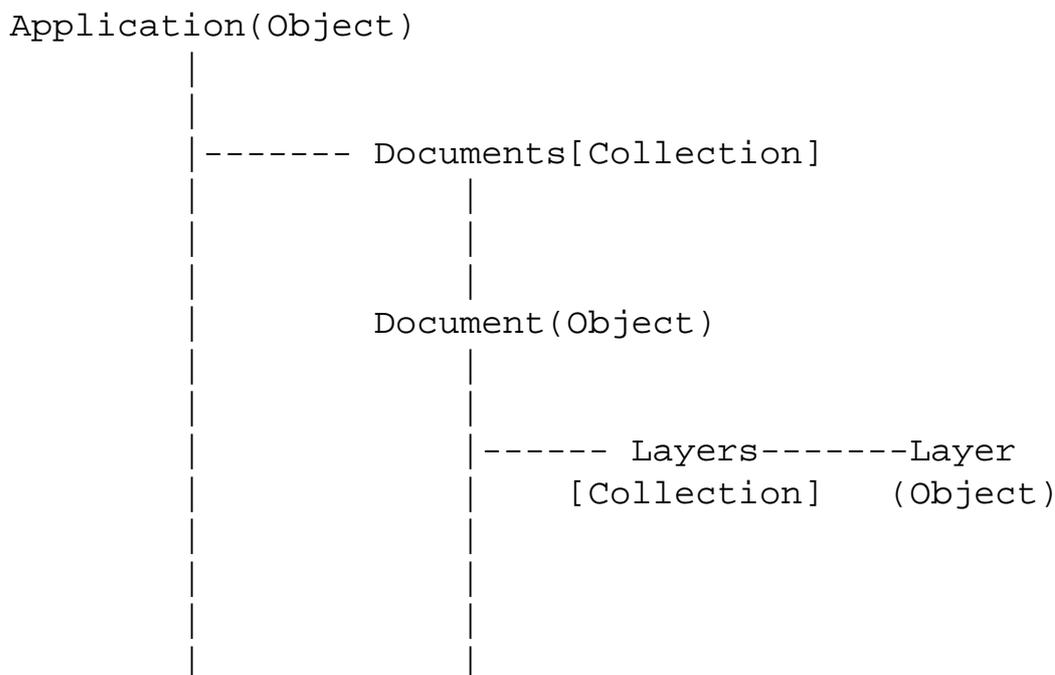
To retrieve a Layer to play with, we first need to access the Layer's Collection.
Type this at the Console prompt :

_$ (vl-load-com)

_$ (setq acadobject (vlax-get-Acad-Object))
#<VLA-OBJECT IAcadApplication 00adc088>

We are now in the Application Object. Now let's sneak into the Document Object :

_$ (setq activedocument (vla-get-activedocument acadobject))
#<VLA-OBJECT IAcadDocument 01945554>

And into the Layers Collection we go!

_$ (setq LayerTable (vla-get-layers activedocument))
#<VLA-OBJECT IAcadLayers 01a070bc>

"But, hang on one second, you jumped from the Application Object straight to the Document Object!!"
Hey, well spotted. I'm glad to see that you're wide awake. OK, I suppose an explanation would be in order.
Let's run a dump on the Application Object :

```
_$ (vlax-dump-object acadobject)
; IAcadApplication: An instance of the AutoCAD application

; Property values:
; ActiveDocument = #<VLA-OBJECT IAcadDocument 01945504>
; Application (RO) = #<VLA-OBJECT IAcadApplication 00adc088>
; Caption (RO) = "AutoCAD 2000 - [Drawing1.dwg]"
; Documents (RO) = #<VLA-OBJECT IAcadDocuments 02d06860>
; FullName (RO) = "C:\\ACAD2000\\acad.exe"
; Height = 776
; LocaleId (RO) = 1033
; MenuBar (RO) = #<VLA-OBJECT IAcadMenuBar 02d096c4>
; MenuGroups (RO) = #<VLA-OBJECT IAcadMenuGroups 015baf2c>
; Name (RO) = "AutoCAD"
; Path (RO) = "C:\\ACAD2000"
; Preferences (RO) = #<VLA-OBJECT IAcadPreferences 015bb9ec>
; StatusId (RO) = ...Indexed contents not shown...
; VBE (RO) = #<VLA-OBJECT VBE 03404a04>
; Version (RO) = "15.0h (Hardware Lock)"
; Visible = -1
; Width = 1032
; WindowLeft = -4
; WindowState = 3
; WindowTop = -4
T
```

Do you see what I see? The Active Document Object is a Property of the Application Object. This means that we can access it directly without having to retrieve it from the Documents Collection. Clever hey?

Still not convinced. OK, let's do it the long way around :

```
_$ (setq acadobject (vlax-get-Acad-Object))
#<VLA-OBJECT IAcadApplication 00adc088>
```

Again we've accessed the Application Object. Now let's get into the Documents Collection :

```
_$ (setq documentcollection (vla-get-documents acadobject))
#<VLA-OBJECT IAcadDocuments 01a03d30>
```

Now, we'll retrieve the Document Object for the drawing

```
_$ (setq thedocument (vla-item documentcollection 0))
#<VLA-OBJECT IAcadDocument 01a6f6a4>
```

And finally we retrieve the Layers Collection.

```
_$ (setq LayerTable (vla-get-layers thedocument))
#<VLA-OBJECT IAcadLayers 02fdca64>
```

But, what happens if we have two or more drawings open and we want to access the Layers Collection of one of the other drawings?
Let's think about this. Where would the inactive documents (your other drawings) be stored?
I would say in the Documents Collection wouldn't you agree?
Let's try this out. First of all open two new drawings, Drawing1.dwg and Drawing2.dwg. Ensure that Drawing1.dwg is the active drawing. Now type this at the Console prompt.

_$ (setq acadobject (vlax-get-Acad-Object))
#<VLA-OBJECT IAcadApplication 00adc088>

Again we've accessed the Application Object. Now let's get into the Documents Collection :

_$ (setq documentcollection (vla-get-documents acadobject))
#<VLA-OBJECT IAcadDocuments 01a03d30>

Now, we'll retrieve the Document Object for the specific inactive drawing

_$ (setq thedocument (vla-item documentcollection "Drawing2.dwg"))
#<VLA-OBJECT IAcadDocument 01a6f6a4>

And finally we retrieve the Layers Collection.

_$ (setq LayerTable (vla-get-layers thedocument))
#<VLA-OBJECT IAcadLayers 02fdca64>

Let's test this out. Let's change the current Layer in Drawing2.dwg to Layer 0. Ensure that Drawing1.dwg is still the active drawing and enter this at the Console prompt :

_$ (vla-put-activelayer thedocument (vla-item LayerTable 0))
nil

Switch to Drawing2.dwg. Layer 0 should have become the current Layer.

To avoid confusing you any further by dealing with multiple workspaces, let's go back to just one active drawing. Close all drawings and open a new drawing. Enter this at the Console prompt :

_$ (setq acadobject (vlax-get-Acad-Object))
#<VLA-OBJECT IAcadApplication 00adc088>
_$ (setq activedocument (vla-get-activedocument acadobject))
#<VLA-OBJECT IAcadDocument 01945554>
_$ (setq LayerTable (vla-get-layers activedocument))
#<VLA-OBJECT IAcadLayers 01a070bc>

No messing about this time! Straight to the Layers Collection.
Right, down to the nitty gritty. First create a Layer in your drawing named "TestLayer".
OK, now let's access "TestLayer" from the Layers Collection :

_$ (setq theLayer (vla-item LayerTable "TestLayer"))
#<VLA-OBJECT IAcadLayer 02fce56c>

Let's list the properties and methods of this Layer :

_$ (vlax-dump-object theLayer T)
; IAcadLayer: A logical grouping of data, similar to transparent acetate overlays on a drawing

; Property values:
; Application (RO) = #<VLA-OBJECT IAcadApplication 00adc088>
; Color = 7

```
; Document (RO) = #<VLA-OBJECT IAcadDocument 01945554>
; Freeze = 0
; Handle (RO) = "957"
; HasExtensionDictionary (RO) = 0
; LayerOn = -1
; Linetype = "CONTINUOUS"
; Lineweight = -3
; Lock = 0
; Name = "TestLayer"
; ObjectID (RO) = 26864120
; ObjectName (RO) = "AcDbLayerTableRecord"
; OwnerID (RO) = 26862608
; PlotStyleName = "Color_12"
; Plottable = -1
; ViewportDefault = 0

; Methods supported:
; Delete ()
; GetExtensionDictionary ()
; GetXData (3)
; SetXData (2)
T
```

To create a new Layer, we simply add it to the Layers Collection :

```
_$ (setq aNewLayer (vla-add LayerTable "NewLayer"))
#<VLA-OBJECT IAcadLayer 02fce7bc>
```

You should have a new Layer in your drawing named "NewLayer".
But, our new Layer has been created with default colour and linetype values namely, 7 and Continuous.
Let's change them :

```
_$ (vla-put-color aNewLayer 2)
nil
```

Changes the colour of our Layer to 2 (yellow).

```
_$ (vla-put-linetype aNewLayer "Dashed2")
nil
```

Changes the linetype of our Layer to "Dashed2". "Dashed 2" of course, must be loaded within our drawing.
Conversely, if you want to find the Colour and Linetype of a particular Layer you would do this :

```
_$ (vla-get-color aNewLayer)
2
```

```
_$ (vla-get-linetype aNewLayer)
"DASHED2"
```

Let's play around with our new Layer. First, change to any other Layer in your drawing.
OK, let's switch our Layer OFF and then back ON :

_$ (vla-put-layeron aNewLayer :vlax-false)
nil

_$ (vla-put-layeron aNewLayer :vlax-true)
nil

Next we'll FREEZE our Layer :

_$ (vla-put-freeze aNewLayer :vlax-true)
nil

And to THAW the Layer :

(vla-put-freeze aNewLayer :vlax-false)
nil

Right, now we'll LOCK it :

_$ (vla-put-Lock aNewLayer :vlax-true)
nil

And now UNLOCK the Layer :

_$ (vla-put-Lock aNewLayer :vlax-false)
nil

Should we now make the Layer UNPLOTTABLE?

_$ (vla-put-plottable aNewLayer :vlax-false)
nil

Now we'll make the Layer PLOTTABLE :

_$ (vla-put-plottable aNewLayer :vlax-true)
nil

Want to change the Layer's LINEWEIGHT? Let's change it to 0,35mm :

_$ (vla-put-LineWeight aNewLayer 35)
nil

Let's change the Lineweight back to DEFAULT :

_$ (vla-put-LineWeight aNewLayer -3)
nil

"ByLwDefault" = -3
"ByBlock" = -2
"ByLayer" = -1
Other Values are : 0, 5, 9, 13, 15, 18, 20, 25, 30, 35, 40, 50, 53, 60, 70, 80, 90, 100, 106, 120, 140, 158, 200,
211.

In AutoCAD, make our new Layer the current Layer, and draw a line.
Now let's delete our new Layer :

_$ (vla-delete aNewLayer)
; error: Automation Error. Object is referenced by other object(s)

Oh, oh. We have an error. Think about it! How can we delete the Layer if it is being referenced by our Line Object?
OK, Let's delete the Line and try again :

_$ (vla-delete aNewLayer)
; error: Automation Error. Object is referenced by other object(s)

What, still an error? That's because the Layer is current and is therefore referenced by the Document Object.
Now make any other Layer current and try for a third time :

_$ (vla-delete aNewLayer)
nil

Hurray, success at last. The Layer is now an "Ex-Layer".

Here's a couple of Layer routines written using Visual Lisp that you may find useful.

Turn All Layers ON :

```
(defun C:Layeron ( / acadDocument theLayers)
(vl-load-com)
(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))
(setq theLayers (vla-get-layers acadDocument))
(vlax-for item theLayers (vlax-put-property item "LayerOn" ':vlax-true) )
(princ)
);defun
(princ)
```

This routine will place a Prefix in front of all Layer names ;and rename them.
Of course, it will not rename Layer "0" or "Defpoints".

```
(prompt "\nType ChLayName to run.........")
(defun C:ChLayName ( / acadDocument theLayers layName pre)
(vl-load-com)
(setq pre (getstring "\nEnter Layer Prefix : "))
(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))
(setq theLayers (vla-get-layers acadDocument))
(vlax-map-collection theLayers 'layer-mod)
(princ)
);defun
```

```
(defun layer-mod (theLayer)
  (setq layName (vlax-get-property theLayer 'Name))
  (if (not (member layName '("0" "Defpoints")))
    (vla-put-Name thelayer (strcat pre layName))
  ) ;if
);defun
(princ)
```

Create a layer using Visual Lisp?

```
;;;Returns a layer object or nil
;;;on creation failure
(defun mLayer (LayerName)
(vl-load-com)
  (setq LayerName
    (vl-catch-all-apply
      'vla-add
      (list
        (vla-get-layers
          (vla-get-activedocument
            (vlax-get-acad-object)
          )
        )
        Layername
      )
    )
  )
  (if (vl-catch-all-error-p LayerName)
    nil
    LayerName
  )
)
```

This routine will return a list of all Layers in the active drawing :

```
(defun C:LayList ( / acadobject activedocument LayerTable thelist)
(vl-load-com)
(setq acadobject (vlax-get-Acad-Object))
(setq activedocument (vla-get-activedocument acadobject))
(setq LayerTable (vla-get-layers activedocument))
(vlax-for each LayerTable
(setq thelist (cons (vla-get-Name each) thelist))
)
(if thelist (reverse thelist))
);defun
(princ)
```

Visual Lisp and Profiles

Profiles are a great way of quickly loading your standard settings and ensuring that drawing office standards are adhered too.

To get to the Profiles in Visual Lisp, we need to reference the PreferencesProfiles Object. Here's an extract from the AutoCAD Object Model :

```
Application (Object)
   |
   |
   |----- Preferences
   |      (Object)
   |         |
   |         |
   |         |
   |         |----- PreferencesProfiles
   |         |           (Object)
   |         |
   |         |
   |         |
```
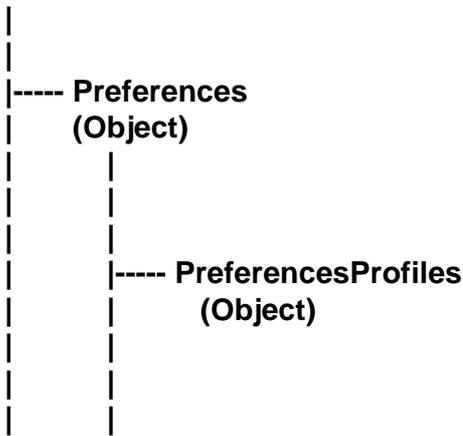
Firstly, we need a reference to the Application Object :

_$ (vl-load-com)

_$ (setq acadobject (vlax-get-Acad-Object))
#<VLA-OBJECT IAcadApplication 00adc088>

And next the Preferences Object :

_$ (setq acadprefs (vla-get-preferences acadobject))
#<VLA-OBJECT IAcadPreferences 02d003cc>

And now a reference to the PreferencesProfiles Object :

$ (setq acadprofiles (vla-get-profiles acadprefs))
#<VLA-OBJECT IAcadPreferencesProfiles 02d003bc>

Let's run a dump on the PreferencesProfile Object :

_$ (vlax-dump-object acadprofiles T)
; IAcadPreferencesProfiles: This object contains the options from the Profiles tab on the Options dialog

; Property values:
; ActiveProfile = "<<Unnamed Profile>>"
; Application (RO) = #<VLA-OBJECT IAcadApplication 00adc088>

; Methods supported:
; CopyProfile (2)
; DeleteProfile (1)
; ExportProfile (2)
; GetAllProfileNames (1)
; ImportProfile (3)

; RenameProfile (2)
; ResetProfile (1)
T

As you can see, this Object has only one Property that really interests us, the Active Profile, but seven Methods that look quite interesting
Let's have a look at the Active Profile Property first :

_$ (setq actprofile (vla-get-ActiveProfile acadprofiles))
"Admin"

This tells us quite clearly that the Active Profile on my system is a profile named "Admin".
But, we would like to have a list of all profile names. For this we would need to use the
GetAllProfilesNames **Method :**

(vlax-invoke-method acadProfiles 'GetAllProfileNames 'thelist)
nil

Let's have a look at "thelist" "

_$ thelist
#<safearray...>

Oh, oh, it's a safearray. We need to convert it :

_$ (vlax-safearray->list thelist)
("Admin" "Eric")

That's better, now we've got a list of all presently loaded profiles.
Let's load a new one :

_$ (setq NewProfile (vlax-invoke-method acadprofiles 'ImportProfile "NDBE51D1"
"c:/Profiles/NDBE51D1.arg" :vlax-true))
nil

Have a look at your "Profiles" under "Options". A new profile should have been Imported.
Let's check that it's there programmically :

_$ (vlax-invoke-method acadProfiles 'GetAllProfileNames 'thelist)
nil
_$ (vlax-safearray->list thelist)
("Admin" "Eric" "NDBE51D1")

OK, I'm happy now, I know it's there.
We still though, need to make it the Active Profile :

_$ (vla-put-ActiveProfile acadProfiles "NDBE51D1")
nil

Let's Export a profile :

_$ (vlax-invoke-method acadProfiles 'ExportProfile "Eric" "c:/profiles/eric.arg")

nil

Have a look in your Profiles directory. You should have a new arg file.
Right, now let's be nasty and delete a profile :

_$ (vlax-invoke-method acadProfiles 'DeleteProfile "NDBE51D1")
; error: Automation Error. Cannot delete a profile that is in use.

Ha, ha, of course we can't delete that profile 'cos it's the Active Profile.
OK, let's try another one :

_$ (vlax-invoke-method acadProfiles 'DeleteProfile "Admin")
nil

Bye, bye, Mr Admin profile.
Let's reset a profile :

_$ (vlax-invoke-method acadProfiles 'ResetProfile "Eric")
nil

Hey, nothing happened! That's because the profile "Eric" is not the Active Profile.
Let's try again :

_$ (vlax-invoke-method acadProfiles 'ResetProfile "NDBE51D1")
nil

That's better, the profile is reloaded.
Let's copy an existing profile to a new profile :

_$ (vlax-invoke-method acadProfiles 'CopyProfile "NDBE51D1" "CopiedProfile")
nil

We now have a new profile named "CopiedProfile". Let's rename it :

_$ (vlax-invoke-method acadProfiles 'RenameProfile "CopiedProfile" "RenamedProfile")
nil

Let's list them again :

_$ (vlax-invoke-method acadProfiles 'GetAllProfileNames 'thelist)
nil
_$ (vlax-safearray->list thelist)
("Eric" "NDBE51D1" "RenamedProfile")

Here's a little application that you may find interesting. It will check the Login Name of the user and automatically load the relevant Profile for that user.

Firstly you need to name your Profiles the same as your Login Name. e.g. if your Login Name is NDBE51D1, your Profile must be named NDBE51D1.ARG. Then you need to store all your user Profiles in the same folder. (This example uses "c:/profiles/").

Irrespective of which user logs in, the specific Profile for that user will be loaded if necessary, and made Active.

Not much in the way of error checking at the moment I'm afraid.....

```
;CODING STARTS HERE
(prompt "\nType LoginProfile to run......")

(vl-load-com)

(defun C:LoginProfile (/ profilename acadprofiles actprofile
                         thelist profilepath)

;retrieve the users login name
(setq profilename (strcase (getvar "LOGINNAME")))

;retrieve a reference to the Profiles
(setq acadprofiles (vla-get-profiles
(vla-get-preferences (vlax-get-Acad-Object))))

;retrieve the Active Profile
(setq actprofile (strcase (vla-get-ActiveProfile acadprofiles)))

;if they are not the same
(if (/= profilename actprofile)

   ;do the following
   (progn

        ;get a list of the loaded profiles
        (vlax-invoke-method acadProfiles 'GetAllProfileNames 'thelist)

         ;convert to a list
        (setq thelist (vlax-safearray->list thelist))

;if the profile is not in the list
(if (not (member profilename thelist))

;do the following
(progn

   ;store the profile file
   (setq profilepath
     (strcat "c:/profiles/" profilename ".arg"))

;if the profile is found
(if (findfile profilepath)

   ;do the following
   (progn

        ;load the profile
        (setq NewProfile (vlax-invoke-method
        acadprofiles 'ImportProfile
        profilename profilepath :vlax-true))

   ;make the profile the Active Profile
   (vla-put-ActiveProfile acadProfiles profilename)
```

```
    );progn

    ;profile file cannot be found - exit
    (prompt (strcat "\nCannot find profile " profilepath))

);if

 );progn

;it is loaded but make the profile the Active Profile
(vla-put-ActiveProfile acadProfiles profilename)

);if


    );progn

            ;We could reload the Profile if we wish.
            ;Just uncomment the next line.
            ;(vlax-invoke-method acadProfiles 'ResetProfile profilename)

);if

    (princ)

);defun

(princ)
;CODING ENDS HERE
```

Visual Lisp and Attributes

When you want to edit attributes in AutoCAD most of us use the "Attedit" command. Firstly, we must select the attribute we would like to edit. Then the "Edit Attribute" dialogue box appears which allows us to add or change the values of our attribute. Personally, I think this dialogue leaves a lot to be desired. You cannot customise it in any way, and it displays all attributes whether you want them or not. As well, if you have a lot of attributes you need to page your way through numerous dialogues before reaching the attribute you want to edit.

In this tutorial we are going to have a look at extracting attribute data from a block, displaying the data in a custom dialogue box, and then updating the attribute data on exit. Right, what do we need to do?

1.   Find the block containing the attribute data. (Why select it when we can get AutoCAD to find it for us.)

2.   Extract the attribute data and display it in a dialogue box.

3.   Allow the user to change the data if he so wishes.

4.   Update the attribute data with the new information entered into the dialogue box.

O.K. fire up AutoCAD and open the drawing Attab-vl.dwg.

Alright, I admit that it's not much of a title block, but it's enough to give you the general idea.

Now, at the command prompt type *(load "Addat-vl")* and then enter.
Now, type *"Addat-vl"* and press enter again.

This dialogue should appear :
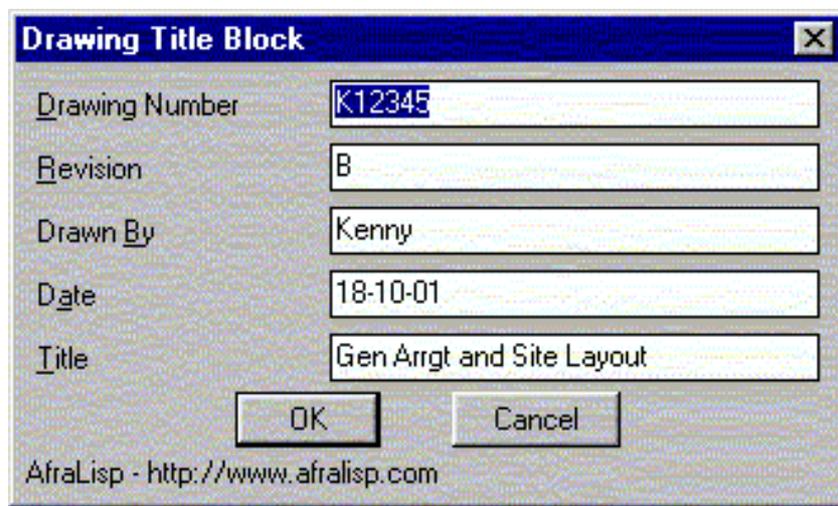
Change some of the data and then press the "OK" button.
The title block data should be updated. Clever hey?

You can expand on this routine as much as you like using the following coding as a template.
Hint : You don't have to display all the attribute data stored in a block. Only display what you want the user to modify. As well, you can split your data over multiple dialogue boxes. eg. One for title block, one for revisions, one for reference drawings, etc. All the data though is contained in one attribute.

Here's the coding, DCL code first :

```
attabvl : dialog {
    label = "Drawing Title Block";

  : edit_box {
    label = "&Drawing Number";
    key = "eb1";
    edit_width = 30;
  }

  : edit_box {
    label = "&Revision";
    key = "eb2";
    edit_width = 30;
  }

  : edit_box {
    label = "Drawn &By";
    key = "eb3";
    edit_width = 30;
  }

  : edit_box {
    label = "D&ate";
    key = "eb4";
```

```
    edit_width = 30;
  }

  : edit_box {
    label = "&Title";
    key = "eb5";
    edit_width = 30;
  }

ok_cancel ;

:text_part {
    label = "AfraLisp - http://www.afralisp.com";
  }


}
```

And now the Visual Lisp coding with plenty of in-line comments to assist you:

```
;CODING STARTS HERE
;
;All Tutorials and Code are provided "as-is" for purposes of instruction and
;utility and may be used by anyone for any purpose entirely at their own risk.
;Please respect the intellectual rights of others.
;All material provided here is unsupported and without warranty of any kind.
;No responsibility will be taken for any direct or indirect consequences
;resulting from or associated with the use of these Tutorials or Code.
;****************************************************************************
; AfraLisp
; http://www.afralisp.com
;  afralisp@afralisp.com
;     afralisp@mweb.com.na
;****************************************************************************
;This application will extract attributes from a block and display them in a
;dialog box. The attributes will then be updated.
;
;Dependencies : Attab-vl.dcl and Attab-vl.dwg are
;required and must be within the AutoCAD search path.
;
;Usage : Open Attab-vl.dwg then load and run Attab-vl.lsp.
;****************************************************************************
;

(prompt "\nATTAB-VL Loaded....Type ATTAB-VL to run.....")

(defun c:attab-vl (/)

;load visual lisp extensions
(vl-load-com)

;retrieve reference to the active document
(setq acadDocument (vla-get-activedocument (vlax-get-acad-object)))
```

```lisp
;retrieve reference to the selection set collection
(setq ssets (vla-get-selectionsets acadDocument))

;check if the selection set exists - $Set
(if (vl-catch-all-error-p (vl-catch-all-apply 'vla-item (list ssets "$Set")))

  ;if it doesn't create a new one
  (setq newsset (vla-add ssets "$Set"))

    ;if it does exist
    (progn

    ;delete it
    (vla-delete (vla-item ssets "$Set"))

    ;then create a new one
    (setq newsset (vla-add ssets "$Set"))

    );progn

);if

;create a single element array - integer
(setq filter_code (vlax-make-safearray vlax-vbinteger '(0 . 0)))

;create a single element array - variant
(setq filter_value (vlax-make-safearray vlax-vbvariant '(0 . 0)))

;filter for name - code 2
(vlax-safearray-fill filter_code '(2))

;filter for block name - attab-info
(vlax-safearray-fill filter_value '("attab-info"))

;filter the drawing for the block
(vla-select newsset acSelectionSetAll nil nil filter_code filter_value)

  ;if the block is found
  (if (>= (vla-get-count newsset) 1)

  ;display the dialog
  (ddisplay)

  ;if the block is not found
  (alert
    "\nIncorrect Drawing Sheet
   \n    Use Manual Edit"
  )

 );if

;finish clean
(princ)

);defun

;;;*******************************************************
;;;

(defun ddisplay (/)
```

```lisp
  ;load the dialog
  (setq dcl_id (load_dialog "attab-vl.dcl"))

  ;check it exists
  (if (not (new_dialog "attabvl" dcl_id))

    (exit)

  );if

;retrieve the block reference
(setq item (vla-item newsset 0))

;retrieve the attributes
(setq theatts (vla-getattributes item))

;convert to a list
(setq attlist (vlax-safearray->list (variant-value theatts)))

;extract the attributes
(mapcar 'set '(theattribute1 theattribute2 theattribute3 theattribute4 theattribute5) attlist)

;extract the text strings from the attributes
(setq eb1 (vla-get-textstring theattribute1)
      eb2 (vla-get-textstring theattribute2)
      eb3 (vla-get-textstring theattribute3)
      eb4 (vla-get-textstring theattribute4)
      eb5 (vla-get-textstring theattribute5)
);setq

  ;put the info into the dialog
  (set_tile "eb1" eb1)
  (set_tile "eb2" eb2)
  (set_tile "eb3" eb3)
  (set_tile "eb4" eb4)
  (set_tile "eb5" eb5)

  ;set the focus to the drawing number
  (mode_tile "eb1" 2)

  ;if cancel selected exit
  (action_tile
    "cancel"
    "(done_dialog) (setq userclick nil)"
  )

  ;if OK selected, retrieve the tile values
  (action_tile
    "accept"
    (strcat
      "(progn (setq eb1a (get_tile \"eb1\"))"
      "(setq eb2a (get_tile \"eb2\"))"
      "(setq eb3a (get_tile \"eb3\"))"
      "(setq eb4a (get_tile \"eb4\"))"
      "(setq eb5a (get_tile \"eb5\"))"
      " (done_dialog)(setq userclick T))"
    )
```

```lisp
  )

  ;start the dialog
  (start_dialog)

  ;unload the dialog
  (unload_dialog dcl_id)


  ;if OK was selected
  (if userclick

;do the following
(progn

;update the attribute textstrings
  (vla-put-textstring theattribute1 eb1a)
(vla-put-textstring theattribute2 eb2a)
(vla-put-textstring theattribute3 eb3a)
(vla-put-textstring theattribute4 eb4a)
(vla-put-textstring theattribute5 eb5a)

;update the block
(vla-update newsset)

  ;regen the drawing
      (command "REGEN")

);progn

  );if

);defun

;;;****************************************************

;load clean
(princ)

;;;****************************************************
;
;CODING ENDS HERE
```

Please note that there is no error checking in this routine and I have left all variables as global to assist you in checking their values whilst you are analyzing the code.

In this tutorial we're going to create an application that extracts attributes from a block and displays these attributes within a dialog box. The dialog will be created "on-the-fly" and the number of attribute edit boxes will be determined by the number of attributes.

We will be using a lot of Visual Lisp coding within this routine as I also want to demonstrate one method of extracting and updating attributes using Visual Lisp.

No further explanation is necessary as the coding is well commented :

```lisp
;;;DCLATT.LSP
;This program is for demonstration and tutorial purposes only.
-----------------------------------------------
;This program, using Visual Lisp will extract attributes from
;a block and display them in a dialog box.
;The dialog box will be created "on the fly" with the relevant
;number of edit boxes ;to suit the number of attributes within
;the block.
;The new attribute data will then be retrieved from the dialog
;and the block updated.
;Written by Kenny Ramage - May 2002
;afralisp@mweb.com.na
;http://www.afralisp.com
-----------------------------------------------
;Usage :
;Load by typing (load "DCLATT") at the command prompt.
;Type DCLATT at the command prompt to run.
;Select any block containing attributes.
;Replace any of the values in the text boxes to updated
;the attributes.
-----------------------------------------------
;Dependencies : None that I know of.
;Not much in the way of error checking I'm afraid.
-----------------------------------------------
;Limitations : Will only edit a certain number of attributes.
;System dependent.
;I don't recommend more than 10.
;I've had up to 14 on my display.
-----------------------------------------------
-----------------------------------------------

(prompt "\nType DCLATT to run.....")

(defun c:dclatt ( / theblock thelist n taglist
                   txtlist lg fname fn nu dcl_id l relist)

;load the visual lisp extensions
(vl-load-com)
```

```
;get the entity and entity name
(setq theblock (car (entsel)))

;convert to vl object
(setq theblock (vlax-ename->vla-object theblock))

;check if it's a block
(if (= (vlax-get-property theblock 'ObjectName)
        "AcDbBlockReference")

        ;if it is, do the following
        (progn

                ;check if it has attributes
                (if (= (vlax-get-property theblock
                    'HasAttributes) :vlax-true)

                    ;if it has attributes, do the following
                    (progn

                        ;get the attributes
                        (getatt theblock)

                        ;create the dialog
                        (create_dialog)

                        ;run the dialog
                        (run_the_dialog)

                        ;update the attributes
                        (upatt)

                    );progn

                        ;No attributes, inform the user
                        (alert "This Block has No Attributes!!
                                - Please try again.")
                );if

        );progn

        ;it's not a block, inform the user
        (alert "This is not a Block!! - Please try again.")

);if

 (princ)

);defun

-----------------------
```

```
(defun getatt (enam)

;retrieve the attributes
(setq thelist (vlax-safearray->list
                 (variant-value
                      (vla-getattributes enam))))

        ;process each attribute
        (foreach n thelist

           ;get the tag attribute data
           (setq taglist (cons (vla-get-tagString n) taglist)

           ;get the text attribute data
                   txtlist (cons (vla-get-textString n) txtlist)

              ;how many attributes?
              lg (length taglist)

           );setq
        );foreach

        ;reverse the lists
        (setq taglist (reverse taglist)
              txtlist (reverse txtlist))

  );defun
------------------------
(defun create_dialog ()

;create a temp DCL file
(setq fname (vl-filename-mktemp "dcl.dcl"))

;open it to write
(setq fn (open fname "w"))

;write the dialog header coding
(write-line "temp : dialog { label = \"Edit Attributes\";" fn)

;reset the incremental control number
(setq nu 0)

;start the loop to create the edit boxes
(repeat lg

;create the edit boxes
(write-line ": edit_box {" fn)
(setq l (strcat "\"" "eb" (itoa nu) "\"" ";"))
(write-line (strcat "key = " l)  fn)
(setq l (nth nu taglist))
```

```
(write-line (strcat "label = " "\"" l "\"" ";") fn)
(setq l (nth nu txtlist))
(write-line (strcat "value = " "\"" l "\"" ";") fn)
(write-line "alignment = centered; edit_width = 20; }" fn)

;increment the counter
(setq nu (1+ nu))

);repeat

;ok and cancel button
(write-line "ok_only; }" fn)

;close the temp DCL file
(close fn)

);defun
------------------------------

(defun run_the_dialog ()

;load the dialog file and definition
(setq dcl_id (load_dialog fname))
     (if (not (new_dialog "temp" dcl_id))
         (exit )
     );if

(mode_tile "eb0" 2)

;if the OK button is selected
(action_tile "accept" "(retatt)")


;start the dialog
(start_dialog)

;unload the dialog
(unload_dialog dcl_id)

;delete the temp DCL file
(vl-file-delete fname)

);defun
--------------------------

(defun retatt ()

;reset the increment counter
(setq nu 0)

start the loop
(repeat lg
```

```
          ;retrieve the tile value
          (setq l (get_tile (strcat "eb" (itoa nu))))

          ;add it to the list
          (setq relist (cons l relist))

          ;increment the counter
          (setq nu (1+ nu))

);repeat

(setq relist (reverse relist))

;close the dialog
(done_dialog)

);defun
----------------------------------------

(defun upatt ()

;reset the increment counter
(setq nu 0)

;start the loop
(repeat lg

  ;update the attribute
  (vla-put-textstring (nth nu thelist) (nth nu relist))

  ;increment the counter
  (setq nu (1+ nu))

);repeat

;update the block
(vla-update theblock)

);defun

--------------------------
;clean loading
(princ)
--------------------------
;End of ATTDCL.LSP
--------------------------
```

Now you need to create 2 or three blocks containing a varying number of attributes.

Here's what mine look like :

Load and run "DCLATT.LSP" and select "Attribute No 1".
**You dialog should appear looking like this with three attribute edit boxes :**
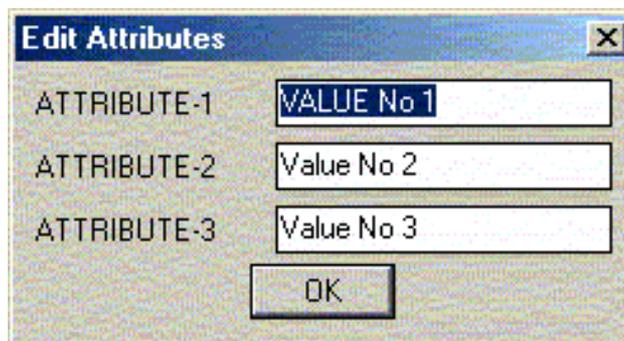
Now run the program again, but this time select "Attribute No 2".
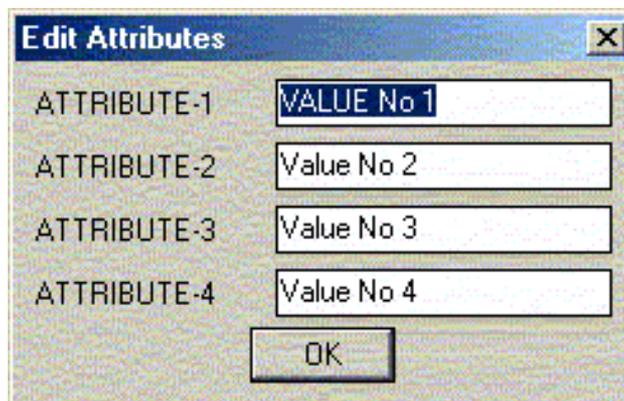
The dialog will appear, but this time with four attribute edit boxes.
Clever hey?

Loading VBA Files

There are two AutoCAD functions that can you would use to Load and Run VBA Applications namely, VBALOAD and VBARUN. In a menu file you would use them like this :

[Test]^C^C^C^P-vbaload test.dvb -vbarun Module1.MyTest

Or, in an AutoLisp routine, you would write something like this :

(command "vbaload" "test.dvb")
(command "-vbarun" "Module1.MyTest")

The VBALOAD function has one serious flaw though!
If a VBA application is already loaded, and you run VBALOAD again, you get an error message. Try it out :

Command: -vbaload
Initializing VBA System...
Open VBA Project: test.dvb

Now try and load it again.

Command: -vbaload
Open VBA Project: test.dvb

You should get an error message :

"File already loaded d:/drawings/test.dvb"

This is where Visual Lisp come into play.
The function (VL-VBALOAD) behaves much like the command VBALOAD. You need to supply the file name of a project or DVB file. The complete file name should be provided along with the path and DVB extension. For example, if you want to load a project named MyProject.dvb in the C:\MyWork\ folder, the (VL-VBALOAD) function call would appear as follows.

(VL-VBALOAD "C:/MyWork/MyProject.DVB")

You should note a couple of things right away. Visual LISP makes use of forward slashes when separating folder or directory names. Also, the parentheses are required and the extension DVB is needed for the project to be properly located.

Unlike the VBALOAD command, this function will not generate an error if the project has already been loaded into the current drawing environment. Thus, programs can proceed smoothly by just calling the load function and then calling the run function without concern about the project already being loaded. Another interesting feature is that the Enable Macros/Virus Warning message does not appear when you use the Visual LISP approach.

Therefore, your menu macro :

[Test]^C^C^C^P-vbaload test.dvb -vbarun MyTest

*c*an be replaced with the following one:

[Test]^C^C^C^P(vl-vbaload "test.dvb")(vl-vbarun "MyTest")

And of course, your AutoLisp coding should be replaced with this :

(vl-vbaload "test.dvb")
(vl-vbarun "MyTest")

Here's a little function that you could load at startup to help you locate, load and run VBA files:

;CODING START HERE

(defun VBA-LOADIT (ProjName Macro)

 (if (findfile ProjName)

 (progn

(vl-vbaload ProjName)

(vl-vbarun Macro)

 );progn

 );if

(princ)

);defun

(princ)

;CODING ENDS HERE

Syntax : *(vbaloadit "dvb-file" "macro")*

Example : *(vbaloadit "test.dvb" "MyTest")*

You must keep some other considerations in mind when using (VL-VBALOAD) and (VL-VBARUN). For example, after you invoke the (VL-VBARUN) function, the Visual LISP function will continue to run and can (will) interfere with the VBA interface if you try to do too much. On the other hand, there are some distinct advantages to using the Visual LISP approach to loading and launching VBA macros instead of the command-line versions when programming a menu- or toolbar-based interface.
One thing to note is that the VBARUN is not a subroutine. That is, program execution will not be handed to the VBA macro and the Visual LISP routine suspended as if it were running a function. Instead, the Visual LISP function will continue to run as the VBA macro starts. The best thing to do is simply finish the Visual LISP function as quickly as possible and let the VBA macro run the command interface from that point forward. If you want to return to a Visual LISP function after running the VBA code, then use the SendCommand method attached to the Document object in VBA. When you are ready to hand control back to Visual LISP, call the function directly (remember to wrap parentheses around the command start up for direct launches of Visual LISP functions). When you use this approach, the VBA program should end and allow the Visual LISP function to proceed without interference. Similar to starting the VBA macro in the first place, when you send commands to the AutoCAD document from VBA, they will be run

along with the VBA and sometimes this can result in confusion at the user level as the two try to take turns. Note that you can pass parameters from VBA to the Visual LISP function by sending them as part of the command stream. They will need to be converted to strings first, then sent to the Visual LISP function as part of the function start up from the Send Command method.

NOTE : *Sorry, but due to additions to the Object Model, this next section will only work in AutoCAD 2002 :-(*

Want to know what Projects are loaded in your drawing?
Type this at the console prompt :

```
_$ (vl-load-com)

_$ (setq oApp (vlax-get-acad-object))
#<VLA-OBJECT IAcadApplication 00ac8928>

_$ (setq oVbe (vlax-get oapp "VBE"))
#<VLA-OBJECT VBE 020b9c18>

_$ (vlax-dump-object oVBE T)

; VBE: nil

; Property values:
; ActiveCodePane = nil
; ActiveVBProject = #<VLA-OBJECT _VBProject 020ba620>
; ActiveWindow (RO) = nil
; CodePanes (RO) = #<VLA-OBJECT _CodePanes 00b1c2e0>
; CommandBars (RO) = #<VLA-OBJECT _CommandBars 030b2a24>
; Events (RO) = #<VLA-OBJECT Events 020b9c94>
; MainWindow (RO) = #<VLA-OBJECT Window 020b8ce8>
; SelectedVBComponent (RO) = #<VLA-OBJECT _VBComponent 020ba748>
; VBProjects (RO) = #<VLA-OBJECT _VBProjects 020b9c4c>
; Version (RO) = "5.00"
; Windows (RO) = #<VLA-OBJECT _Windows 020b9d18>

; No methods
T
```

I presume you can see what I see? A "VBProjects" property.
Now that's interesting! But how do we extract the loaded Projects?
Load and run this small routine.

```
;CODING STARTS HERE

(defun Gvba ( /oApp oVBE oProjs N Nams oProj)

(vl-load-com) ;requires automation links

(if (and

  ;Drill down to the Projects object
  (setq oApp (vlax-get-acad-object))
  (setq oVBE (vla-get-vbe oApp))
  (setq oProjs (vlax-get oVBE "VBProjects"))
```

```
)

;Loop through Projects object
(repeat (setq N (vla-get-count oProjs))

;get the item at position N
(setq oProj (vla-item oProjs N)

;get the name property,
;add it to the list.
Nams (cons
(list
(vlax-get oProj "Name")
(vlax-get oProj "FileName")
) Nams) N (1- N)))
)

; return list of names
Nams

);defun

;CODING ENDS HERE
```

You should have a list of Projects in the variable "Nams".

And, would you like to Unload all Projects within your drawing? Try this :

```
;CODING STARTS HERE

(defun C:UNLOADALLVBA ( / VBAProjs VBAProj)

(setq VBAProjs (Gvba))

(foreach VBAProj VBAProjs

(command "_VBAUNLOAD" (cadr VBAProj)))

)

;CODING ENDS HERE
```

Visual Lisp - Directories and Files

Using plain old AutoLisp, we really only have two functions dealing with files and directories namely, the (findfile) function, and the (getfiled) function. Both of these functions are useful but limited in there scope.

Visual Lisp has introduced a whole host of new functions specifically designed to use with files and directories. In this tutorial we're going to have a look at a few of them. Firstly, we'll have a look at probably the most powerful Visual Lisp function for dealing with files and directories, the (vl-directory-files) function. Fire up AutoCAD, open the Visual Lisp Editor, then type this at the console prompt :

_$ (vl-load-com)

_$ (vl-directory-files)
("." ".." "X3426.dwg" "addshort.dvb" "ADMENU.DVB" "afralispURL.dvb" "afraLOGO.dwg" "another-dump.lsp" "area.zip" "arrowkeyview.dvb" "ATTAB.DCL" "ATTAB.DWG" "ATTAB.LSP")

This will return a list containing every file and sub-directory within your current directory.
Let's have a look at the syntax of (vl-directory-files) and a few other Visual Lisp File/Directory functions.

VL-DIRECTORY-FILES

Lists all files in a given directory

(vl-directory-files [directory pattern directories])

Arguments :

directory

- A string naming the directory to collect files for; if nil or absent, vl-directory-files uses the current directory.

pattern

- A string containing a DOS pattern for the file name; if nil or absent, vl-directory-files assumes "*.*"

directories

- An integer that indicates whether the returned list should include directory names. Specify one of the following :

  -1   List directories only.
   0   List files and directories (the default).
   1   List files only.

Return Values

- A list of file and path names, or nil, if no files match the specified pattern.

Let's try it again, but this time with some arguments. Type this at the console prompt :

_$ (vl-directory-files "d:/drawings" "*.dwg")
("X3426.dwg" "afraLOGO.dwg" "ATTAB.DWG" "Drawing1.dwg" "Drawing4.dwg" "is handsome.dwg" "tem5.dwg" "X3374.dwg" "X3375.dwg" "dwgdata.dwg" "Drawing2.dwg" "X3359.dwg" "tblock.dwg" "kenny.dwg" "Adaptor.dwg" "Drg-1.dwg" "drg-2.dwg" "attab-vl.dwg" "matlist.dwg")

This will return a list of all drawings residing in the directory "d:/drawings".
Let's try something else :

_$ (vl-directory-files "d:/drawings" "*.lsp")
("another-dump.lsp" "ATTAB.LSP" "BAREA.LSP" "bick.lsp" "acad.lsp" "mted.lsp" "circle-react.lsp" "clay.lsp" "DC-Delete.lsp" "endPlot.lsp")

This, of course, will return a list of all AutoLisp files.
Now let's get clever. Let's try and get a list of just the subdirectories :

_$ (vl-directory-files "d:/drawings1" nil -1)
("." ".." "Purge-Export" "Office" "3D" "Extext" "VBA" "DrawingExfiles")

Easy hey. Now let's have a look at some of the other Visual Lisp file handling functions.

VL-FILE-COPY

Copies or appends the contents of one file to another file

(vl-file-copy source-file destination-file [append])

Copy or append the contents of one file to another file. The vl-file-copy function will not overwrite an existing file, only append to it.

Arguments :

source-file

- **A string naming the file to be copied. If you do not specify a full path name, vl-file-copy looks in the AutoCAD start-up directory.**

destination-file

- **A string naming the destination file. If you do not specify a path name, vl-file-copy writes to the AutoCAD start-up directory.**

append

- **If specified and not nil, source-file is appended to destination-file (that is, copied to the end of the destination file).**

Return Values

■ **An integer, if the copy was successful, otherwise nil.**
Some typical reasons for returning nil are:

source-file **is not readable**
source-file **is a directory**
append? **is absent or nil and destination-file exists**
destination-file **cannot be opened for output (that is, it is an illegal file name or a write-protected file)**
source-file **is the same as destination-file**

Examples

Copy autoexec.bat to newauto.bat:

_$ (vl-file-copy "c:/autoexec.bat" "c:/newauto.bat")
1417

Copy test.bat to newauto.bat:

_$ (vl-file-copy "c:/test.bat" "c:/newauto.bat")
nil

The copy fails because newauto.bat already exists, and the append argument was not specified. Repeat the previous command, but specify append:

_$ (vl-file-copy "c:/test.bat" "c:/newauto.bat" T)
185

The copy is successful because T was specified for the append argument.

VL-FILE-DELETE

Deletes a file

(vl-file-delete filename)

Arguments :

filename

■ **A string containing the name of the file to be deleted. If you do not specify a full path name, vl-file-delete searches the AutoCAD start-up directory.**

Return Values

■ **T, if successful, nil if delete failed.**

Examples

Delete newauto.bat:

_$ (vl-file-delete "newauto.bat")
nil

Nothing was deleted because there is no newauto.bat file in the AutoCAD start-up directory.
Delete the newauto.bat file in the c:\ directory:

_$ (vl-file-delete "c:/newauto.bat")
T

The delete was successful because the full path name identified an existing file.

VL-FILE-DIRECTORY-P

Determines if a file name refers to a directory

(vl-file-directory-p filename)

Arguments :

filename

- A string containing a file name. If you do not specify a full path name, vl-file-directory-p searches only the AutoCAD start-up directory.

Return Values

- T, if filename is the name of a directory, nil if it is not.

Examples

_$ (vl-file-directory-p "sample")
T

_$ (vl-file-directory-p "yinyang")
nil

_$ (vl-file-directory-p "c:/program files/acad2000")
T

_$ (vl-file-directory-p "c:/program files/acad2000/vlisp/yinyang.lsp")
nil

VL-FILE-RENAME

Renames a file

(vl-file-rename old-filename new-filename)

Arguments :

old-filename

■ A string containing the name of the file you want to rename. If you do not specify a full path name, vl-file-rename looks in the AutoCAD start-up directory.

new-filename

■ A string containing the new name to be assigned to the file.
NOTE If you do not specify a path name, vl-file-rename writes the renamed file to the AutoCAD start-up directory.

Return Values

■ T, if renaming completed successfully, nil if renaming failed.

Examples

_$ (vl-file-rename "c:/newauto.bat" "c:/myauto.bat")
T

VL-FILE-SYSTIME

Returns last modification time of the specified file

(vl-file-systime filename)

Arguments :

filename

■ A string containing the name of the file to be checked.

Return Values

■ A list containing the modification date and time, or nil, if the file is not found.
The list returned contains the following elements :

year
month
day-of-week
day-of-month
hours
minutes
seconds

Note that Monday is day 1 of day-of-week, Tuesday is day 2, etc.

Examples

_$ (vl-file-systime "c:/program files/acad2000/sample/visuallisp/yinyang.lsp")
(1998 4 3 8 10 6 52 0)

The returned value shows that the file was last modified in 1998, in the 4th month of the year (April), the 3rd day of the week (Wednesday), on the 10th day of the month, at 6:52:0.

VL-FILENAME-BASE

Returns the name of a file, after stripping out the directory path and extension

(vl-filename-base filename)

Arguments :

filename

    ■ A string containing a file name. The vl-filename-base function does not check to see if the file exists.

Return Values

    ■ A string containing filename in uppercase, with any directory and extension stripped from the name.

Examples

_$ (vl-filename-base "c:\\acadwin\\acad.exe")
"ACAD"

_$ (vl-filename-base "c:\\acadwin")
"ACADWIN"

VL-FILENAME-DIRECTORY

Returns the directory path of a file, after stripping out the name and extension

(vl-filename-directory filename)

Arguments :

filename

    ■ A string containing a complete file name, including the path. The vl-filename-directory function does not check to see if the specified file exists. Slashes (/) and backslashes (\) are accepted as directory delimiters.

Return Values

    ■ A string containing the directory portion of filename, in uppercase.

Examples

_$ (vl-filename-directory "c:\\acadwin\\acad.exe")
"C:\\ACADWIN"

_$ (vl-filename-directory "acad.exe")

VL-FILENAME-EXTENSION

Returns the extension from a file name, after stripping out the rest of the name

(vl-filename-extension filename)

Arguments :

filename

- A string containing a file name, including the extension. The vl-filename-extension function does not check to see if the specified file exists.

Return Values

- A string containing the extension of filename. The returned string starts with a period (.) and is in uppercase. If filename does not contain an extension, vl-filename-extension returns nil.

Examples

_$ (vl-filename-extension "c:\\acadwin\\acad.exe")
".EXE"

_$ (vl-filename-extension "c:\\acadwin\\acad")
nil

VL-FILENAME-MAKETEMP

Calculates a unique file name to be used for a temporary file

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

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).

Return Values

- A string containing a file name, in the following format :
  directory\base<XXX><.extension>

  where:
  base is up to 5 characters, taken from pattern
  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.

Examples

_$ (vl-filename-mktemp)
"C:\\TMP\\$VL~~004"

_$ (vl-filename-mktemp "myapp.del")
"C:\\TMP\MYAPP005.DEL"

_$ (vl-filename-mktemp "c:\\acadwin\\myapp.del")
"C:\\ACADWIN\MYAPP006.DEL"

_$ (vl-filename-mktemp "c:\\acadwin\\myapp.del")
"C:\\ACADWIN\MYAPP007.DEL"

_$ (vl-filename-mktemp "myapp" "c:\\acadwin")
"C:\\ACADWIN\MYAPP008"

_$ (vl-filename-mktemp "myapp" "c:\\acadwin" ".del")
*"C:\\ACADWIN\MYAPP00A.DEL"*

OK, enough of this theoretical nonsense! Let's do something practical.
How do you fancy creating your own personalised file list box?
You do? Great! I'll see you on the next page.

Right, are you ready to create your own file list box? OK here we go. First I'll give you a wee peek at what our dialog will look like :
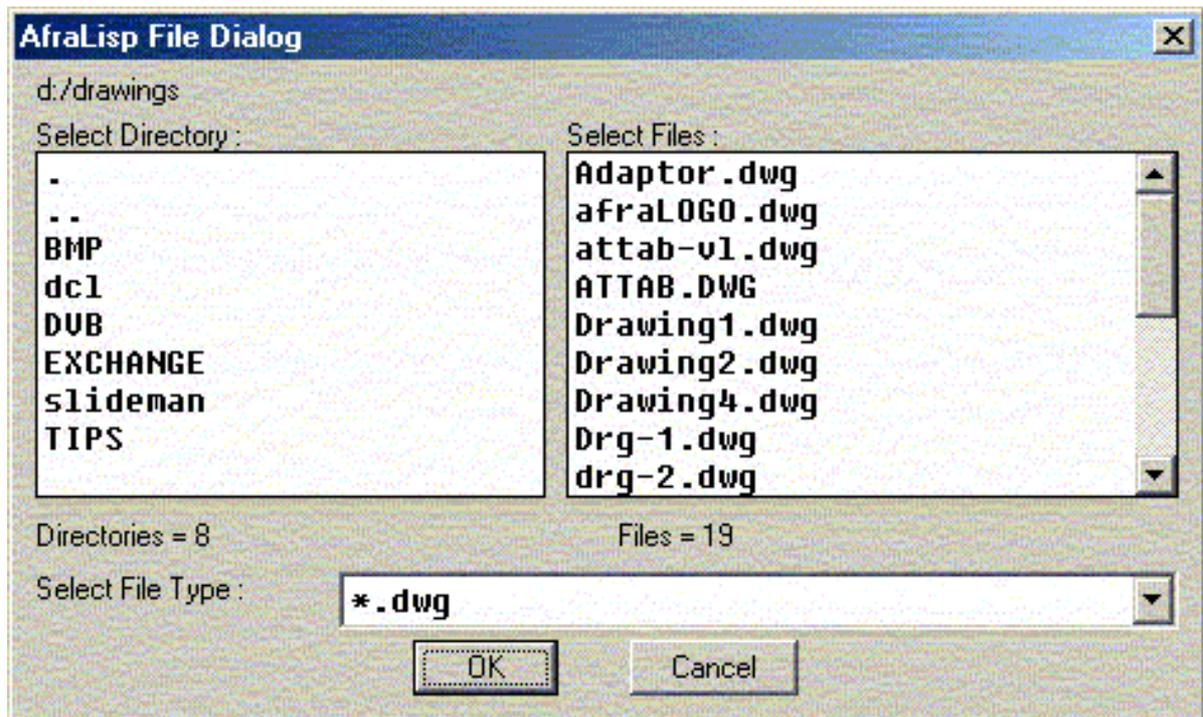
Looks good hey!
To run this function you must pass it two arguments :

■ **A string containing a directory path.**
■ **A list of file types.**

Syntax : *(fileselect [directory file_ types])*

**Example : (fileselect "d:/drawings" '("*.dwg" "*.lsp" "*.dvb"))*

Return : *A list of selected files.*

Oh, by the way, you CAN select multiple files.

The first thing we need to do is write a bit of DCL to create our File Dialog. Copy and paste this into Notepad and save it as "AfraFile.dcl".

```
FILES : dialog {
   label="AfraLisp File Dialog";

   : text {
   key="CDIR";
   }

   : row {

   : list_box {
   key="DIR";
```

```
    label="Select Directory :";
    width=25;
    fixed_width_font = true;
    }

    : list_box {
    key="FIL";
    label="Select Files :";
    width = 30;
    tabs = "20 31 40";
    multiple_select = true;
    fixed_width_font = true;
    }

    }

    : row {

    : text {
    key="DIRS";
    }

    : text {
    key="FILS";
    }

    }

    : popup_list {
    key="EXT";
    label="Select File Type :";
    fixed_width_font = true;
    }

    ok_cancel;

}
```

And now the AutoLisp Coding. Save this as AfraFile.lsp :

```
;CODING STARTS HERE
;;Syntax : (fileselect "d:/drawings" '("*.dwg" "*.lsp" "*.dvb"))

(defun FileSelect (Dir Pat)
(setq DH (load_dialog "afrafiles"))
  (if (and DH (new_dialog "FILES" DH))
    (progn
      (setq iExt 0)
      (Refresh_Display)
      (start_list "EXT")
      (mapcar 'add_list Pat)
      (end_list)
      ;
      (action_tile "DIR" "(new_dir $value)")
      (action_tile "EXT" "(new_mask $value)")
```

```
    (action_tile "FIL" "(picked $value)")
    ;
    (if (= (start_dialog) 0)
(setq File_List nil)
    )
    (unload_dialog DH)
  )
 )
  File_List
)
;------------------------------------------
(defun Refresh_Display ()
 (start_list "FIL")
 (end_list)
 (set_tile "CDIR" "Working...")
 (setq FL (VL-Directory-Files
        ;Dir Pat 1)
        Dir (nth iExt Pat) 1)
DR (VL-Directory-Files
    Dir nil -1)
FL (VL-Sort FL 'str_compare)
DR (VL-Sort DR 'str_compare)
)
 (start_list "DIR")
 (mapcar 'add_list DR)
 (end_list)
 (start_list "FIL")
 (if Show_the_details
   (mapcar
    '(lambda (F)
(setq Dt (VL-File-SysTime
    (strcat Dir F))
     F1 (if Dt
    (strcat
     F
     "\t"
     (itoa_f (nth 1 Dt) 2)
     "/"
     (itoa_f (nth 3 Dt) 2)
     "/"
     (itoa_f (nth 0 Dt) 4)
     "\t"
     (itoa_f (nth 4 Dt) 2)
     ":"
     (itoa_f (nth 5 Dt) 2)
     ":"
     (itoa_f (nth 6 Dt) 2)
     )
    (strcat F "\t\t")
  )
     Sz (VL-File-Size (strcat Dir F))
     F1 (strcat
    F1
    "\t"
    (rtos Sz 2 0))
     )
    (add_list F1))
```

```
  FL)
    (mapcar 'add_list FL)
  )
  (end_list)
  (set_tile "DIRS"
    (strcat
      "Directories = "
      (itoa (length DR))))
  (set_tile "FILS"
    (strcat
      "Files = "
      (itoa (length FL))))
  (set_tile "CDIR" Dir)
  )

(defun New_Dir (Pth)
  (setq Pth (nth (atoi Pth) DR))
  (cond
    ((= Pth ".")
     nil
     )
    ((= Pth "..") ;;back up a directory
     ;;remove directory name up one
     (setq L (1- (strlen Dir))
  Dir (substr Dir 1 L)
   )
     (while (/= (substr Dir L 1) "/")
       (setq L (1- L)))
     (setq Dir (substr Dir 1 L))
    )
    ('T
     (setq Dir (strcat Dir Pth "/"))
     )
  )
  (Refresh_Display)
)
;--------------------------------------------
;; Call back function to handle new file mask
;; selection by the user.
;;
(defun New_Mask (ll)
  (setq iExt (atoi ll))
  (Refresh_Display)
)
;
;--------------------------------------------
;; Call back function for saving the selected
;; file list in the variable FILE_LIST.
;;
(defun Picked (val / V)
  (setq val (read (strcat "(" Val ")")))
File_List
(mapcar '(lambda (V)
    (strcat
      Dir
      (nth V FL)))
Val)
```

```
)
)
;;-------------------------------------------
;; Convert integer to padded ASCII string
;;
(defun Itoa_F (I Digs)
  (setq I (itoa I))
  (while (< (strlen I) Digs)
    (setq I (strcat "0" I)))
  I
  )
;;-------------------------------------------
(defun Str_Compare (T1 T2)
  (< (strcase T1)
     (strcase T2)))

(princ)
;;-------------------------------------------
;CODING ENDS HERE
```

I just like to thank Bill Kramer from whom I "stole" a lot of this coding from.
(Shush, don't say anything as he doesn't know yet!!!)

Compiling AutoLisp Files

Before we start with various methods of compiling AutoLisp files, let's have a look at the different file types we'll encounter during this Tutorial :

- **_lsp_**   **AutoLisp Program Source file**
- **_dcl_**   **Contains definitions of AutoCAD Dialog Boxes**
- **_fas_**   **Compiled AutoLisp Program**
- **_vlx_**   **Executable Visual Lisp Module**
- **_prv_**   **Defines the files and options used to build a _vlx_ module**

Each time you load AutoLISP source code, the code is translated into instructions the computer understands (executable code). The advantage of having source code translated each time you load it is that you can make a change and immediately try it out. This is useful for quickly testing new code, and for debugging that code.
Once you are sure your program is working correctly, translating AutoLISP source code each time it loads is time-consuming. VLISP provides a compiler that generates executable machine code files from your source files. These executable files are known as **FAS** files. Because the executable files contain only machine-readable code, the source code you spent weeks or months developing remains hidden even if you distribute your program to thousands of users. Even strings and symbol names are encrypted by the VLISP file compiler.

VLISP also provides features for packaging more complex AutoLISP applications into VLISP executable (**VLX**) files. **VLX** files can include additional resources files, such as **VBA**, **DCL** and **TXT** files, as well as compiled AutoLISP code.

So, let's try and give you a digest of what we've just discussed :

- **If you have a single AutoLisp file, compile it as a _fas_ file.**
- **If you have an AutoLisp file with dependencies such as _DCL_ or _TXT_ files, compile it as an Executable Visual Lisp Module _vlx_ file.**

Let's start off this Tutorial by compiling a single AutoLisp file.
Copy and paste this into Notepad and save it as "Slot.lsp" :

```
(defun C:SLOT (/ oldsnap diam lngth pt1 pt2 pt3 pt4 pt5 pt6)

(setvar "CMDECHO" 0)
(setvar "BLIPMODE" 0)
(setq oldsnap (getvar "OSMODE"))

(setq diam (getdist "\nSlot Diameter : ")
      lngth (getdist "\nSlot Length : "))

(while
   (setq pt1 (getpoint "\nInsertion point: "))
   (setvar "OSMODE" 0)
   (setq pt2 (polar pt1 0.0 (/ (- lngth diam) 2.0))
         pt3 (polar pt2 (/ pi 2.0) (/ diam 4.0))
         pt4 (polar pt3 pi (- lngth diam))
```

```
            pt5 (polar pt4 (* pi 1.5) (/ diam 2.0))
            pt6 (polar pt5 0.0 (- lngth diam)))

    (command "PLINE" pt3 "W" (/ diam 2.0) "" pt4
             "ARC" pt5 "LINE" pt6 "ARC" "CLOSE")
    (setvar "OSMODE" oldsnap)
  );while
  (princ)
);defun
(princ)
```

Now fire up AutoCAD and open the Visual Lisp Editor. Type this at the console prompt :

_$ (vlisp-compile 'st "slot.lsp")
T

Have look at the Build Output window :

; (COMPILE-FILES st (D:/drawings/slot.lsp))
[Analyzing file "D:/drawings/slot.lsp"]
..
[COMPILING D:/drawings/slot.lsp]
;;C:SLOT
;
[FASDUMPING object format -> "D:/drawings/slot.fas"]
; Compilation complete.

During compilation, the compiler prints function names and various messages about each stage of compilation. The first stage is syntax and lexical checking of the source code. If the compiler encounters errors, it issues messages and halts the compilation process. The compiler issues warnings if it encounters expressions it considers dangerous, such as redefining existing AutoLISP functions or assigning new values to protected symbols. If the compiler displays warning or error messages, you can view and edit the source code that caused these messages by double-clicking on the message in the Build Output window.

If compilation is successful, as in the example above, the Build Output window displays the name of the compiled output file. This file should be located in the same directory as your AutoLisp source file. Let's have a look at the syntax for (vlisp-compile) :

VLISP-COMPILE

Compiles AutoLISP source code into a FAS file

(vlisp-compile 'mode filename [out-filename])

NOTE The Visual LISP IDE must be open in order for vlisp-compile to work.

Arguments :

- ■ **mode**

  The compiler mode, which can be one of the following symbols:
  st   Standard build mode
  lsm Optimize and link indirectly
  lsa Optimize and link directly

- ■ **filename**

  A string identifying the AutoLISP source file. If the source file is in the AutoCAD Support File Search Path, you can omit the path when specifying the file name. If you omit the file extension, .lsp is assumed.

- ■ **out-filename**

  A string identifying the compiled output file. If you do not specify an output file, vlisp-compile names the output with the same name as the input file, but replaces the extension with .fas. Note that if you specify an output file name but do not specify a path name for either the input or the output file, vlisp-compile places the output file in the AutoCAD install directory.

Return Values :

- ■ **T, if compilation is successful, nil otherwise.**

Examples

Assuming that slot.lsp resides in a directory that is in the AutoCAD Support File Search Path, the following command compiles this program :

_$ (vlisp-compile 'st "slot.lsp")
T

The output file is named slot.fas and resides in the same directory as the source file.
The following command compiles slot.lsp and names the output file Slot-1.fas :

(vlisp-compile 'st "slot.lsp" "slot-1.fas")

Note that the output file from the previous command resides in the AutoCAD install directory, not the directory where slot.lsp resides. The following command compiles slot.lsp and directs the output file to the c:\my documents directory :

(vlisp-compile 'st "slot.lsp" "c:/my documents/slot-1.fas)

This last example identifies the full path of the file to be compiled :

(vlisp-compile 'st "c:/program files/acad2000/Sample/slot.lsp")

The output file from this command is named slot.fas and resides in the same directory as the input file.


Next, we'll be having a look at creating compiled AutoLisp files using multiple AutoLisp files and dependency files such as DCL files.

Visual Lisp provides you with the ability to create a single, standalone executable module for your application. This module incorporates all your application's compiled files, and can include *DCL*, *DVB*, and other files that your application may need. Executable Visual Lisp modules are known as *VLX* files, and are stored in files named with a *.vlx* extension.

A Make Application wizard guides you through the application building process in Visual Lisp. The result of this process is a Make file, which is often referred to by its file extension, *.prv*. The Make file contains all the instructions Visual Lisp needs to build the application executable.

To test this out, I've provided an AutoLisp Application that consists of an AutoLisp file, and a DCL file. From these 2 files we will compile one *vlx* executable module. (Just click **here** to download the source files). Unzip these files and save them in any directory within your AutoCAD Support Path.

OK, fire up AutoCAD and open the Visual Lisp Editor. Now select "Files" - "Make Application" - "New Application" from the pulldown menu :
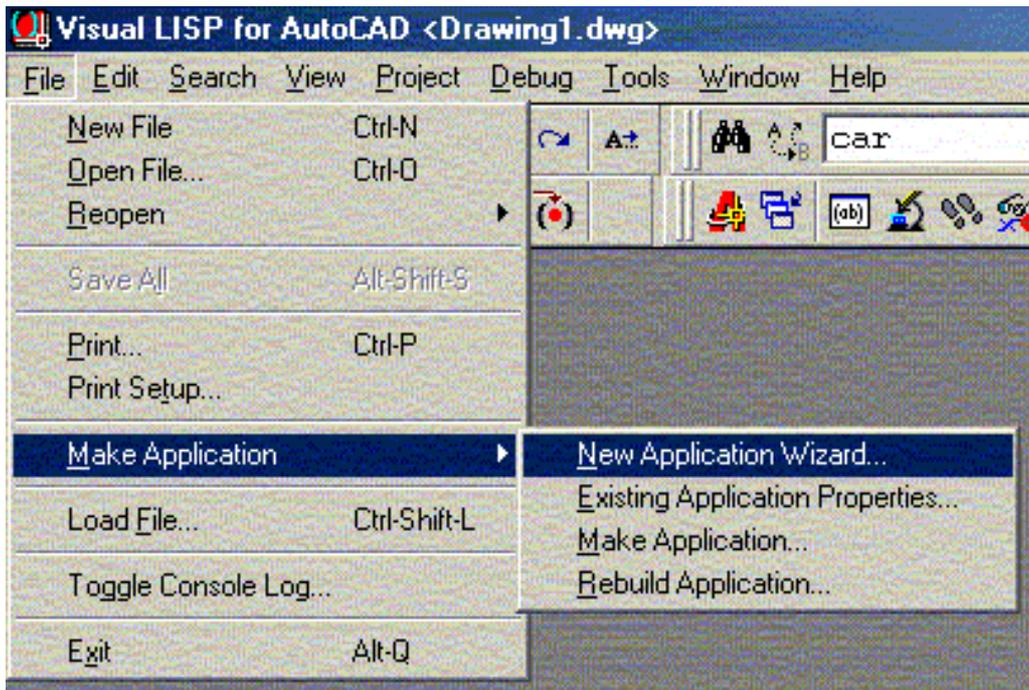
This dialog will appear :
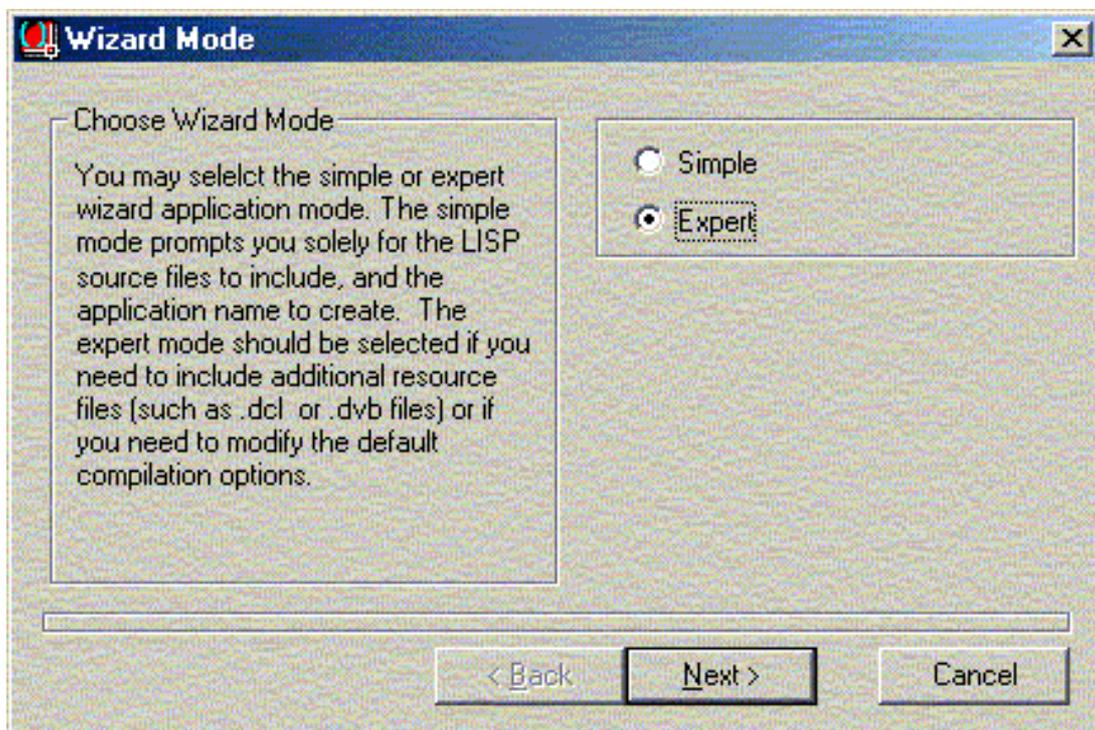
Select "Expert" mode as we want to compile multiple file types, and then press "Next" :
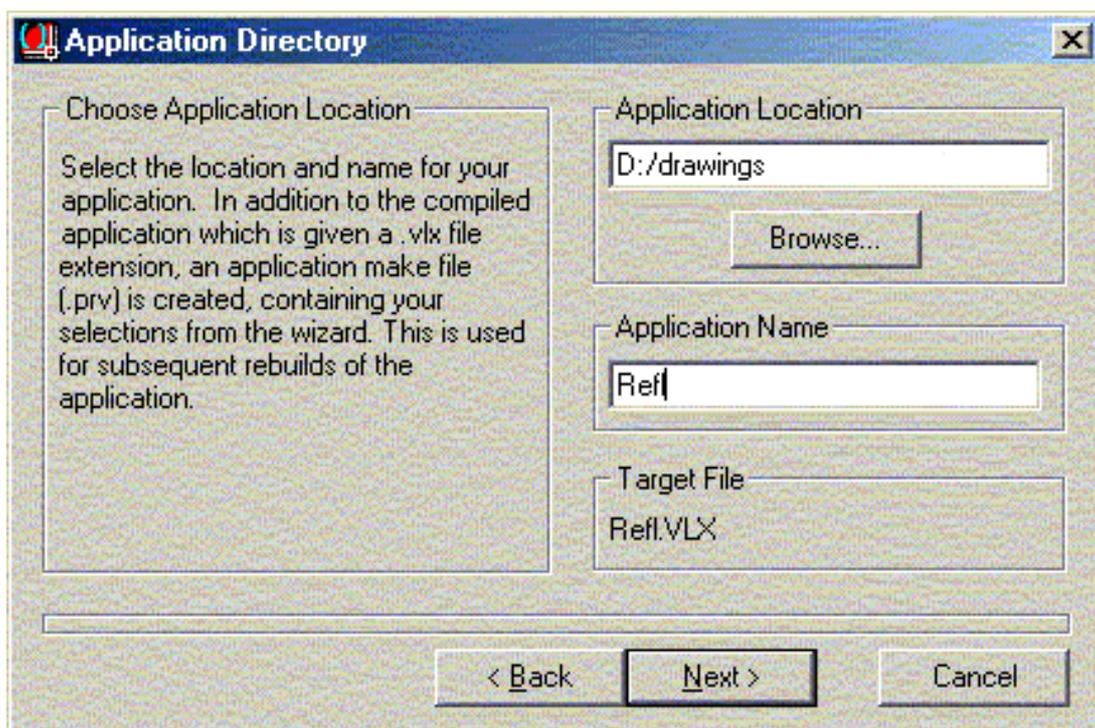
Enter the path to the directory you would like to store the files in. Select the same directory that you stored the AutoLisp source files.
Then, give your application a name. It does not have to be the same as your AutoLisp source file, but to spare any confusion, let's keep it the same.
The select "Next" :
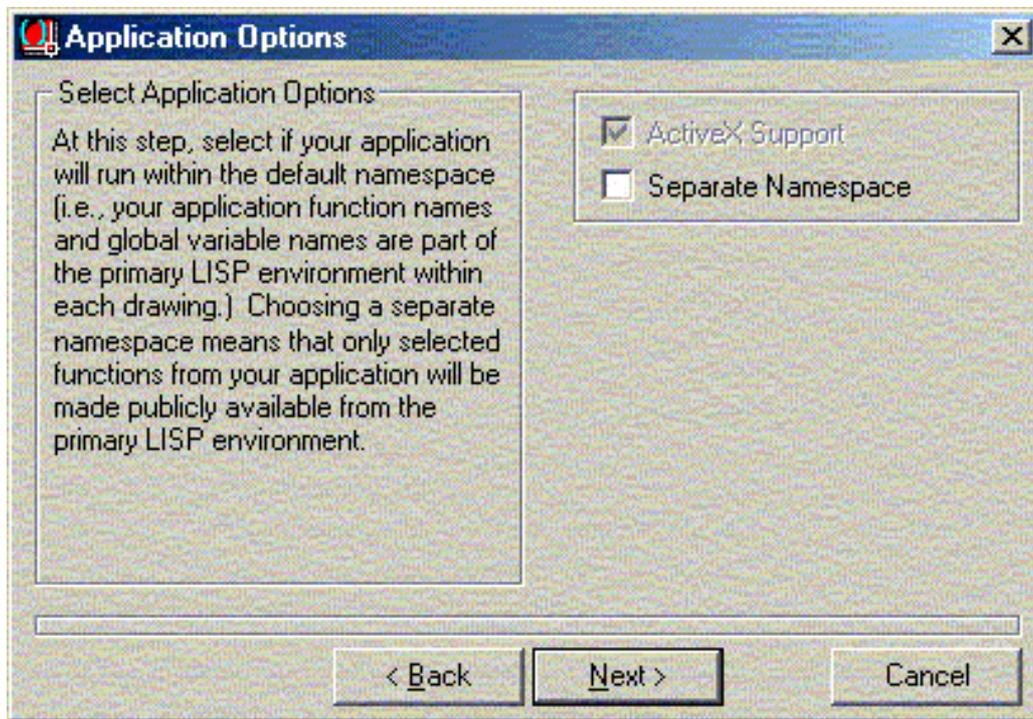
No, we do not want our application to run in a separate namespace.
Just select "Next" :
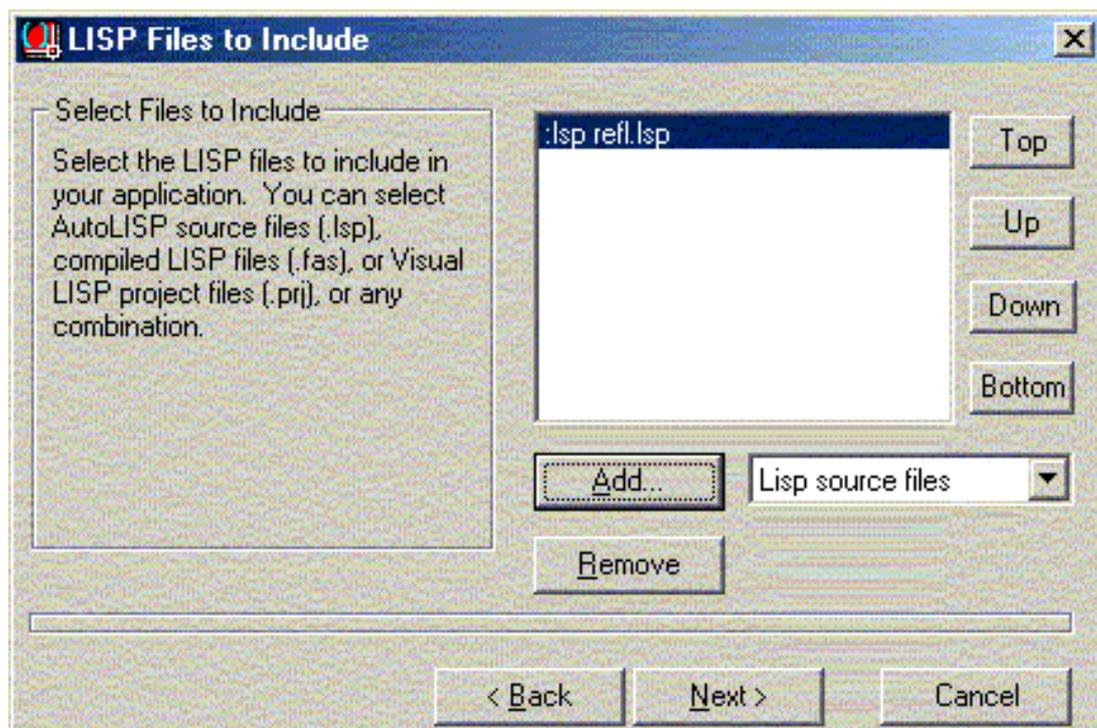
Now we need to select our AutoLisp source file. Select "refl.lsp" then press "Next" :
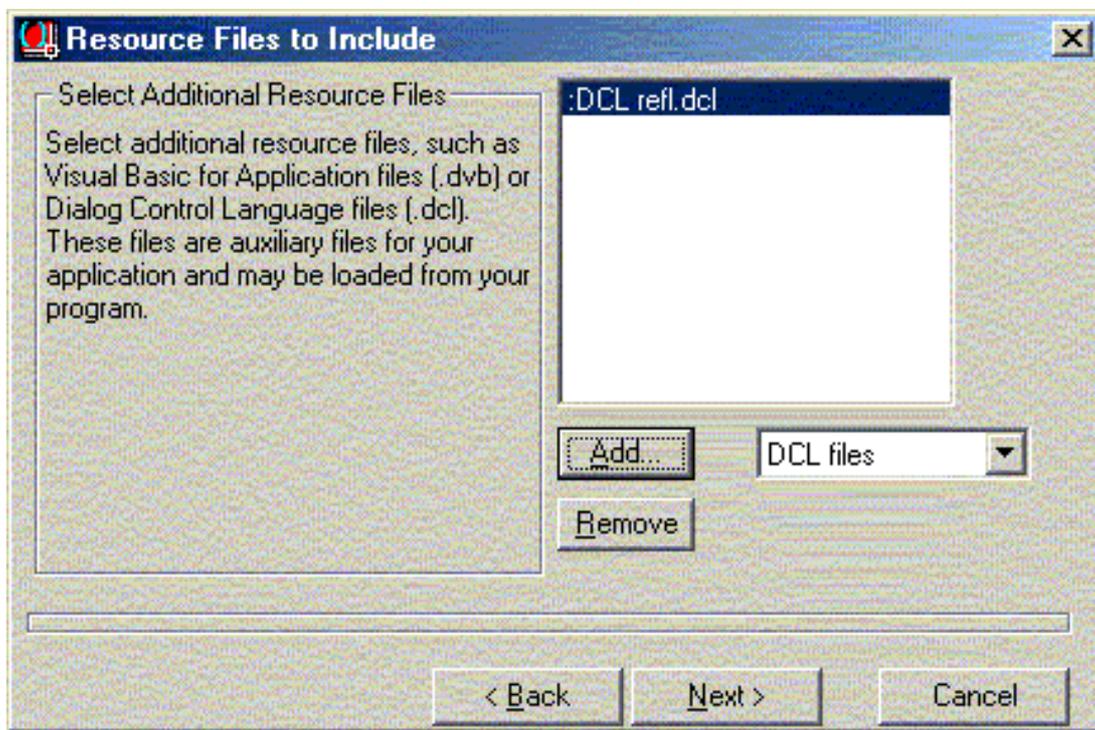
In this dialog, we select and add any dependency files that our application needs to run correctly. Select "refl.dcl" and then "Next" :
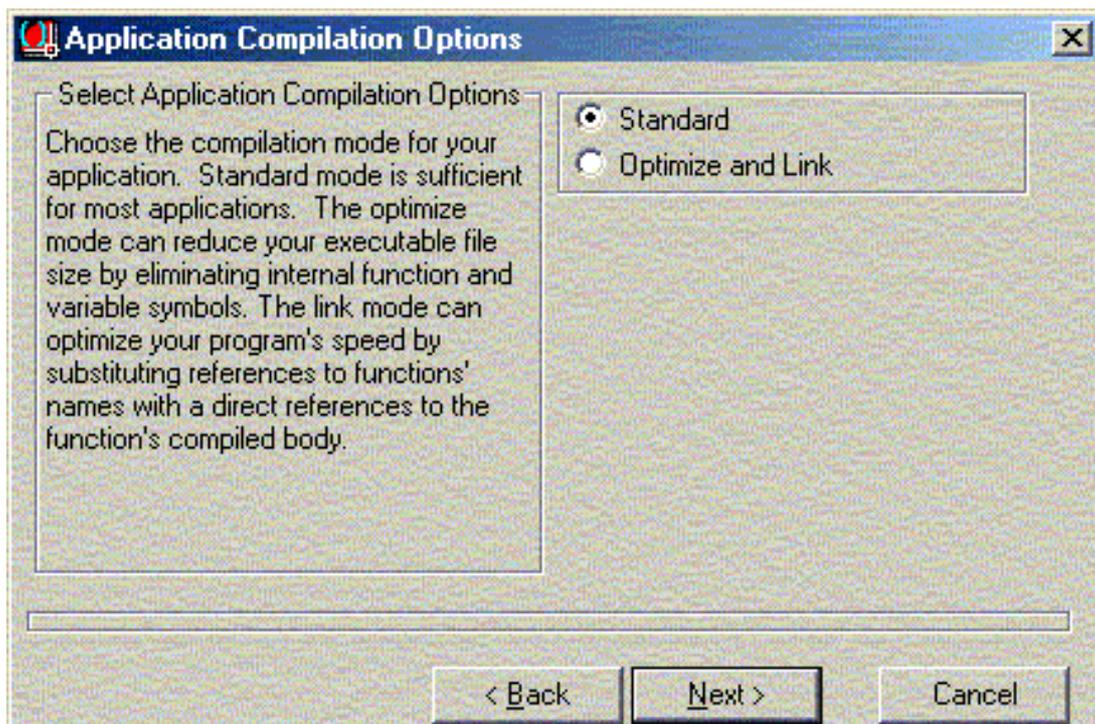
No need to confuse you now, just go with the "Standard" option. Select "Next" :
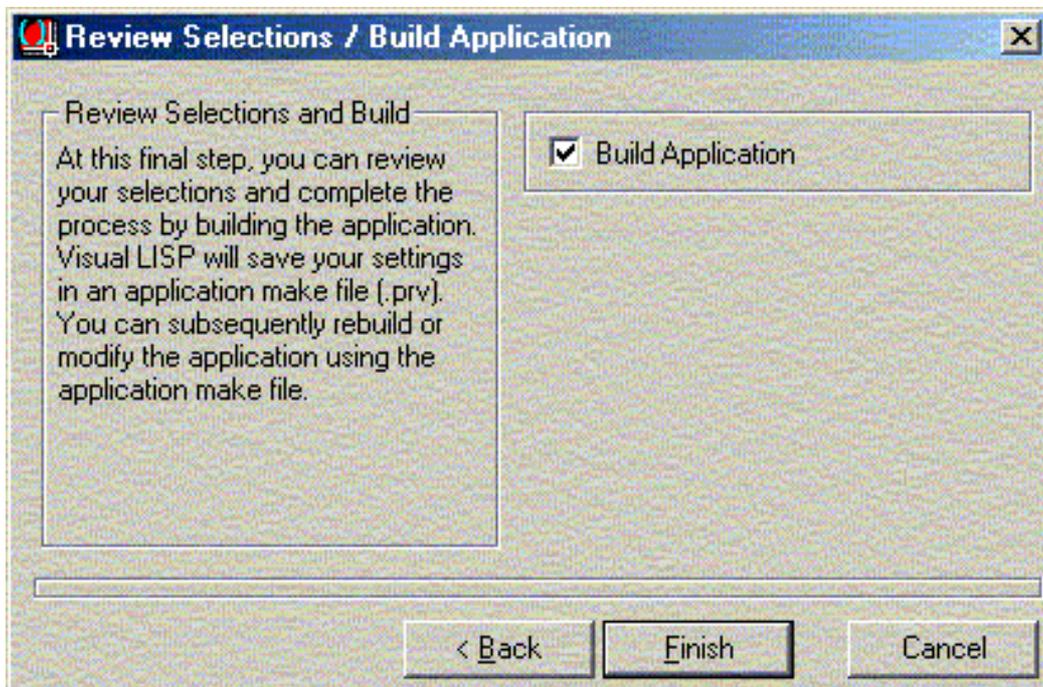
OK, that's us about done. To build your compiled application, just select "Finish".

**VLISP executes instructions in a Make file to build an application. Output messages from this process appear in two VLISP windows: the Build Output window and the Console window. The Build Output window contains messages relating to any compilation of AutoLISP source code into *.fas* files. In a successful compile, the output looks like the following :**

; (COMPILE-FILES st (D:/drawings/refl.lsp))
[Analyzing file "D:/drawings/refl.lsp"]
..
......
[COMPILING D:/drawings/refl.lsp]
;
;;C:REFL
;CCC
;;MKLIST
;;SPINBAR
;;INITERR
;;TRAP
;;RESET
;
[FASDUMPING object format -> "D:/drawings/refl.fas"]
; Compilation complete.

The compiler messages identify the following items:

- **The name and directory path of the source files being compiled.**
- **The functions defined in the source file. Seven functions are identified: C:REFL, CCC, MKLIST, SPINBAR, INITERR, TRAP and RESET.**
- **The name and path of the output *.fas* files.**

The VLISP Console window displays messages relating to the creation of the application executable, the *.vlx* file. If the Make Application process succeeds, the Console window displays the path and file name of the *.vlx*, as in the following example:

Have a look in the directory where you stored the source files. You should now have 5 "*refl*" files :

- *refl.lsp* - AutoLisp source file
- *refl.dcl* - DCL source file
- *refl.fas* - Compiled AutoLisp Program
- *refl.vlx* - Executable Visual Lisp Module
- *refl.prv* - Application Make file.

You can now distribute "*refl.vlx*" as a compiled application.

Note!!! You *CANNOT* edit a *vlx* file. These 5 files need to be kept in a safe location if you intend to edit or revise your *vlx* file.

Loading Compiled AutoLisp Applications :

Loading compiled applications is exactly the same as for normal AutoLisp functions :

(load "refl")

Here's a couple of things to keep in mind :

If you do not specify a file extension, load first looks for a file with the name you specified (for example, "*refl*"), and an extension of *.vlx*. If no *.vlx* file is found, load searches next for a *.fas* file, and finally, if no *.fas* file is found, load searches for a *.lsp* file.

Tip of the Day :

To aid you in the process of maintaining multiple-file applications, VLISP provides a construct called a Project. A VLISP Project contains a list of AutoLISP source files, and a set of rules on how to compile the files.
Using the Project definition, VLISP can do the following :

- Check which .lsp files in your application have changed, and automatically recompile only the modified files. This procedure is known as a Make procedure.
- Simplify access to source files by listing all source files associated with a project, making them accessible with a single-click.
- Help you find code fragments by searching for strings when you do not know which source files contain the text you're looking for. VLISP limits the search to files included in your project.
- Optimize compiled code by directly linking the corresponding parts of multiple source files.

Have a look at the Visual Lisp Help for further information on Projects.

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
```

Thanks to David Stein from whom I "borrowed" this listing.

Visual Lisp and Polylines

Dealing with polylines using straight forward AutoLisp can be quite a pain. But, believe it or not, using Visual Lisp they are a breeze to modify and manipulate. Let's have a look shall we?

First of all, fire up AutoCAD and draw a polyline but do not close it :
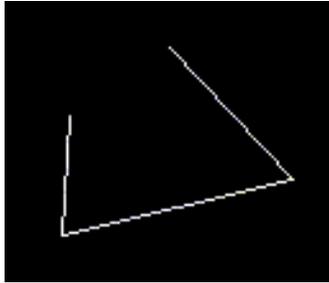
Now, type the following in the Visual Lisp editor :

```
_$ (vl-load-com)

_$ (setq theobj (car (entsel "\nSelect a Polyline: ")))
<Entity name: 14e35f8>

_$ (setq theobj (vlax-ename->vla-object theobj))
#<VLA-OBJECT IAcadLWPolyline 01ea16d4>

_$ (vlax-dump-object theobj T)
; IAcadLWPolyline: AutoCAD Lightweight Polyline Interface

; Property values:
; Application (RO) = #<VLA-OBJECT IAcadApplication 00ac8928>
; Area (RO) = 16178.7
; Closed = 0
; Color = 256
; ConstantWidth = 0.0
; Coordinate = ...Indexed contents not shown...
; Coordinates = (540.98 557.623 640.815 449.587 453.624 403.879 ... )
; Document (RO) = #<VLA-OBJECT IAcadDocument 00ec89b4>
; Elevation = 0.0
; Handle (RO) = "957"
; HasExtensionDictionary (RO) = 0
; Hyperlinks (RO) = #<VLA-OBJECT IAcadHyperlinks 01ea1ec4>
; Layer = "7"
; Linetype = "BYLAYER"
; LinetypeGeneration = 0
; LinetypeScale = 1.0
; Lineweight = -1
; Normal = (0.0 0.0 1.0)
; ObjectID (RO) = 21902840
; ObjectName (RO) = "AcDbPolyline"
; OwnerID (RO) = 21901504
; PlotStyleName = "ByLayer"
; Thickness = 0.0
; Visible = -1

; Methods supported:
```

This is a listing of all the Properties and Methods belonging to our polyline object.
Let's close the polyline:

_$ (vla-put-closed theobj :vlax-true)
nil
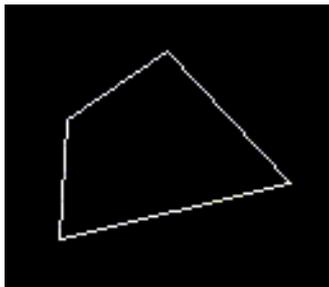
Now let's change the width of all the segments :

_$ (vla-put-ConstantWidth theobj 2.0)
nil
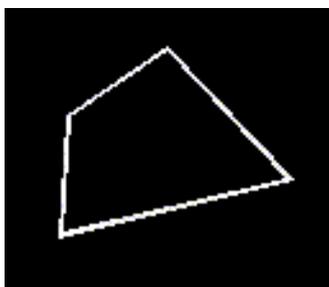
Let's "bulge" the third segment :

_$ (vla-setbulge theobj 2 0.5)
nil
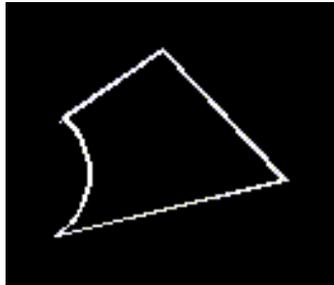
Let's change the starting and ending width of the fourth segment :

_$ (vla-setwidth theobj 3 10.0 0.0)
nil

Let's get the area :

_$ (vla-get-area theobj)
14505.9

Now, we'll make it invisible :

_$ (vla-put-visible theobj :vlax-false)
nil

See it's gone. Let's bring it back :

_$ (vla-put-visible theobj :vlax-true)
nil

Now we'll explode it :

_$ (vla-explode theobj)
#<variant 8201 ...>

And delete the original :

_$ (vla-delete theobj)
nil

We are left with an exploded copy of our polyline :
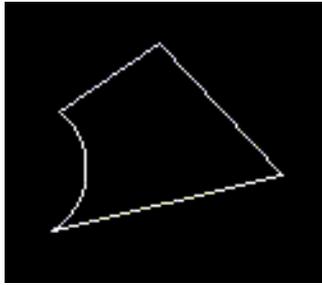
Right, let's have a look at extracting some information from a polyline.
This program will extract the X and Y coordinates from any polyline.

```
(prompt "\nType \"VL-POLY\" to run........")

(defun c:vl-poly ( / theobj thelist n xval yval fname fn)

;load the visual lisp extensions
(vl-load-com)

;get the entity and entity name
(setq theobj (car (entsel "\nSelect a Polyline: ")))

;convert to vl object
(setq theobj (vlax-ename->vla-object theobj))

;check if it's a polyline
(if (= (vlax-get-property theobj 'ObjectName) "AcDbPolyline")

;if it is, do the following
(progn

;retrieve the coordinates
(setq thelist (vlax-get-property theobj 'coordinates))

;convert to a list
(setq thelist (vlax-safearray->list  (variant-value thelist)))

;zero the counter
(setq n 0)

;create a text file
(setq fname "coord.txt")

;open it to write
(setq fn (open fname "w"))

;write the header
(write-line "PolyLine X and Y Coordinates" fn)

;underline the header
(write-line "**************************************" fn)

;start the loop
(repeat (/ (length thelist) 2)
```

```
;get the x coordinate
(setq xval (rtos (nth n thelist)))

;increase the counter
(setq n (1+ n))

;get the y coordinate
(setq yval (rtos (nth n thelist)))

;write the x coordinate to the file
(write-line  (strcat "X-Value : " xval)  fn)

;write the x coordinate to the file
(write-line  (strcat "Y-Value : " yval)  fn)

;add a seperator
(write-line "---------------------------" fn)

;increase the counter
(setq n (1+ n))

);repeat

;close the file
(close fn)

);progn

;it's not a polyline, inform the user
(alert "This is not a Polyline!! - Please try again.")

);if

 (princ)

);defun

;----------------------
;clean loading
(princ)
;-------------------------
;End of VL-POLY.LSP
;-------------------------
```

 Save this as "VL-Poly.lsp" and then load and run it. Select any polyline.

The X and Y coordinates of each vertex will be output and written to a file named "Coord.txt"
It should look something like this :

PolyLine X and Y Coordinates
****

X-Value : 478.6
Y-Value : 622

X-Value : 815.5
Y-Value : 349.9

X-Value : 636.7
Y-Value : 291.7

X-Value : 586.7
Y-Value : 437.1

X-Value : 516
Y-Value : 310.4

X-Value : 349.6
Y-Value : 304.2

In the next section, we'll have a look at creating polylines and adding one or two "bulges".

This is the VBA method to create a lightweight polyline from a list of vertices.

RetVal = object.AddLightweightPolyline(VerticesList)

- *Object* : ModelSpace Collection, PaperSpace Collection, Block.
  The object or objects this method applies to.
- *VerticesList* : Variant (<u>array</u> of doubles)
  The <u>array</u> of 2D OCS coordinates specifying the vertices of the polyline. At least two points (four elements) are required for constructing a lightweight polyline. The <u>array</u> size must be a multiple of 2.
- *RetVal* : LightweightPolyline object.
  The newly created LightweightPolyline object.

In Visual Lisp, the syntax would be as follows :

(vla-addLightweightPolyline Object VerticesList)

and would return *RetVal.*

Let's try this out. First we need a reference to Model Space :

_$ **(setq mspace (vla-get-modelSpace (vla-get-activeDocument (vlax-get-acad-object)))))**
#<VLA-OBJECT IAcadModelSpace 01ea5064>

Get the first point :

_$ **(setq pt1 (getpoint))**
(424.505 252.213 0.0)

Extract the X and Y coordinates

_$ **(setq pt1 (list (car pt1) (cadr pt1)))**
(424.505 252.213)

And now the second point :

_$ **(setq pt2 (getpoint))**
(767.689 518.148 0.0)

Extract the X and Y coordinates :

_$ **(setq pt2 (list (car pt2) (cadr pt2)))**
(767.689 518.148)

Join the the two lists together :

_$ **(setq pt1 (append pt1 pt2))**
(424.505 252.213 767.689 518.148)

Construct a 4 element <u>safearray</u> :

_$ **(setq anarray (vlax-make-safearray vlax-vbDouble '(0 . 3)))**
#<safearray...>

Fill it with our point X and Y values :

_$ (vlax-safearray-fill anarray pt1)
#<safearray...>

Draw the polyline :

_$ (setq myobj (vla-addLightweightPolyline mspace anarray))
#<VLA-OBJECT IAcadLWPolyline 01ea5914>

Let's now have a look how we could apply this to a practical example :

```
;CODING STARTS HERE

(prompt "\nType \"VL-Steel\" to run......")

;set up default rotation angle
(if (= rot nil) (setq rot 0))

;define the function and declare all local variables
(defun C:VL-Steel ( / ptlist oldsnap oldecho oldblip
acaddoc util mspace names sizes dcl_id siza userclick
dlist H B T1 T2 R1 IP IPA P1 ptlisp tmp myobj fname
fn pts)

;load VL functions
(vl-load-com)

;obtain reference to the Active Document
(setq acaddoc (vla-get-activeDocument (vlax-get-acad-object)))

;obtain reference to Utilities
(setq util (vla-get-utility acaddoc))

;obtain reference to Model Space
(setq mspace (vla-get-modelSpace acaddoc))

;store system variables
(setq oldsnap (vla-getvariable acaddoc "OSMODE")
      oldecho  (vla-getvariable acaddoc "CMDECHO")
        oldblip  (vla-getvariable acaddoc "BLIPMODE")
);setq

;switch off system variables
(vla-setvariable acaddoc "CMDECHO" 0)
(vla-setvariable acaddoc "BLIPMODE" 0)

;create list of steel sections for the list box
```

```
(setq names '("100x55x8" "120x64x10" "140x73x13" "160x82x16"
"180x91x19" "200x100x22" "203x133x25" "203x133x30" "254x146x31"
"254x146x37" "254x146x43" ) )

;create list of steel section values
(setq sizes '((100.0 55.0 4.1 5.7 7.0)
(120.0 64.0 4.4 6.3 7.0) (140.0 73.0 4.7 6.9 7.0)
(160.0 82.0 5.0 7.4 9.0) (180.0 91.0 5.3 8.0 9.0)
(200.0 100.0 5.6 8.5 12.0) (203.2 133.4 5.8 7.8 7.6)
(206.8 133.8 6.3 9.6 7.6) (251.5 146.1 6.1 8.6 7.6)
(256.0 146.4 6.4 10.9 7.6) (259.6 147.3 7.3 12.7 7.6)))

;construct the dialog
(create_dialog)

;load the dialog
  (setq dcl_id (load_dialog fname))
  (if (not (new_dialog "ubeam" dcl_id))
    (exit)
  )

;setup the list box
  (start_list "selections")
  (mapcar 'add_list names)
  (end_list)

;default rotation angle
  (set_tile "rot" (rtos rot))

;setup the Cancel button
  (action_tile
    "cancel"
    "(done_dialog) (setq userclick nil)"
  )

;setup the OK button
  (action_tile
    "accept"
    (strcat
      "(progn (setq siza (atof (get_tile \"selections\")))
                (setq rot (atof (get_tile \"rot\")))"
      "(done_dialog) (setq userclick T))" )
  )

;display the dialog
  (start_dialog)

;unload the dialog
  (unload_dialog dcl_id)
```

```lisp
;delete the temp DCL file
  (vl-file-delete fname)

;if the OK button was selected
  (if userclick

  ;do the following
    (progn

    ;retrieve the steel section values
      (setq dlist (nth (fix siza) sizes))

    ;place them into variables
      (mapcar 'set '(H B T1 T2 R1) dlist)

    ;switch on the intersection snap
      (vla-setvariable acaddoc "OSMODE" 32)

    ;get the insertion point
      (setq IP (vla-getpoint util nil "\nInsertion Point : "))

    ;switch off the snaps
      (vla-setvariable acaddoc "OSMODE" 0)

      ;calculate the points and store them in a list
      (setq pts (list
        (setq P1 (vla-polarpoint util IP 0 (/ T1 2)))
        (setq P1 (vla-polarpoint util P1 (DTR 90.0)
                (/ (- H (+ T2 T2 R1 R1)) 2)))
        (setq P1 (vla-polarpoint util P1 (DTR 45.0)
                (sqrt (* R1 R1 2.0))))
        (setq P1 (vla-polarpoint util P1 0
                (/ (- B (+ T1 R1 R1)) 2)))
        (setq P1 (vla-polarpoint util P1 (DTR 90.0) T2))
        (setq P1 (vla-polarpoint util P1 (DTR 180.0) B))
        (setq P1 (vla-polarpoint util P1 (DTR 270.0) T2))
        (setq P1 (vla-polarpoint util P1 0
                (/ (- B (+ T1 R1 R1)) 2)))
        (setq P1 (vla-polarpoint util P1 (DTR 315.0)
                (sqrt (* R1 R1 2.0))))
        (setq P1 (vla-polarpoint util P1 (DTR 270.0)
                (- H (+ T2 T2 R1 R1))))
        (setq P1 (vla-polarpoint util P1 (DTR 225.0)
                (sqrt (* R1 R1 2.0))))
        (setq P1 (vla-polarpoint util P1 (DTR 180.0)
                (/ (- B (+ T1 R1 R1)) 2)))
        (setq P1 (vla-polarpoint util P1 (DTR 270.0) T2))
        (setq P1 (vla-polarpoint util P1 0 B))
        (setq P1 (vla-polarpoint util P1 (DTR 90.0) T2))
        (setq P1 (vla-polarpoint util P1 (DTR 180.0)
```

```
                          (/ (- B (+ T1 R1 R1)) 2)))
          (setq P1 (vla-polarpoint util P1 (DTR 135.0)
                    (sqrt (* R1 R1 2.0)))))
          (setq P1 (vla-polarpoint util IP 0 (/ T1 2)))
            ));setq

            ;extract only the X and Y values of each point list
              (mapcar

                      '(lambda  (pt)

                        ;convert to lists
                    (setq pt (vlax-safearray->list (variant-value pt)))

                        ;X and Y values only
                    (setq ptlist (cons (list (car pt) (cadr pt)) ptlist))

                 );lambda

       pts

       );mapcar

       ;break the point list up into elements
       (setq ptlist (apply 'append ptlist))

       ;create a safearray to store the elements
       (setq tmp (vlax-make-safearray vlax-vbDouble
                 (cons 0 (- (vl-list-length ptlist) 1))))

       ;fill the safearray
       (vlax-safearray-fill tmp ptlist)

       ;draw the steel section
       (setq myobj (vla-addLightweightPolyline mspace tmp))

       ;radius the corners
       (vla-setbulge myobj 1 0.4142)
       (vla-setbulge myobj 7 0.4142)
       (vla-setbulge myobj 9 0.4142)
       (vla-setbulge myobj 15 0.4142)

       ;rotate the object
       (vla-rotate myobj ip (dtr rot))

   );progn

);if

       ;reset system variables
        (vla-setvariable acaddoc "OSMODE" oldsnap)
```

```lisp
            (vla-setvariable acaddoc "CMDECHO" oldecho)
            (vla-setvariable acaddoc "BLIPMODE" oldblip)

          ;release all objects
            (vlax-release-object mspace)
            (vlax-release-object util)
            (vlax-release-object acaddoc)

    ;finish clean
    (princ)

);defun

;------------------------
(defun create_dialog ()

;create a temp DCL file
(setq fname (vl-filename-mktemp "dcl.dcl"))

;open it to write
(setq fn (open fname "w"))

;write the dialog coding
(write-line
"ubeam : dialog {
      label = \"VL-Steel\";
      : list_box {
         label = \"Choose Section :\";
         key = \"selections\";
         allow_accept = true;
         height = 8;
      }
      : edit_box {
        label = \"Rotation Angle :\";
        key = \"rot\";
        edit_limit = 4;
        edit_width = 4;
       }
       spacer;
     ok_cancel ;
     :text_part {
         label = \"Designed and Created\";
      }
     :text_part {
         label = \"by Kenny Ramage\";
      }
     }" fn)

;close the temp DCL file
(close fn)
```

```
);defun
;--------------------------

;convert degrees to radians

(defun DTR (a)

  (* pi (/ a 180))

);defun

;--------------------------------

;load clean
(princ)

;CODING ENDS HERE
```

Save this as "VL-Steel.lsp" and then load and run it. A dialog like this will appear :
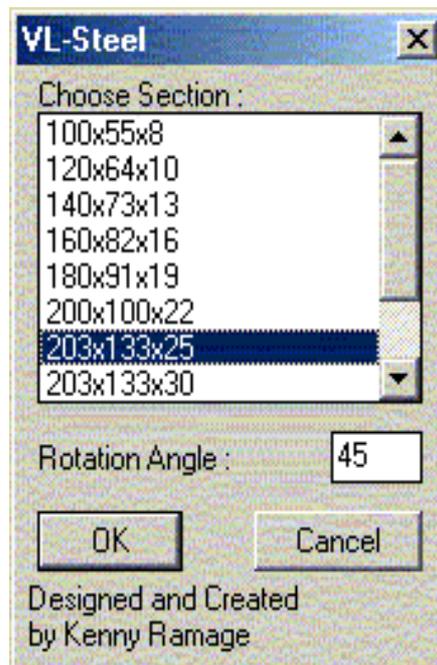
Choose the section size you would like, choose a rotation angle, select an insertion point, and voila, your steel section is drawn using a polyline.

"OK Kenny, now we give in. HOW do you calculate a bulge?"

Bulge = TAN (/ Included Angel 4)

In our case Bulge = TAN (/ 90 4) = 0.4142

Thanks to Stig Madsen for the insight into some of this coding.

Utilities

Whilst looking around the web at other peoples coding, I noticed that nearly everyone still uses the traditional AutoCAD "get" functions to retrieve entities and values.
In this tutorial, we're going to have a look at the Visual Lisp methods that are available to us to achieve the same results. We find these methods in the "Utility" object.

Note! Most of the Visual Lisp "get" methods do not flip to the AutoCAD screen from the Visual Lisp Editor. You need to activate AutoCAD manually and then return to the Visual Lisp Editor.

Let's first have a look at the "standard" way of retrieving points and the drawing a line. Type the following in the console :

(vl-load-com)

_$ (setq acaddoc (vla-get-activedocument (vlax-get-acad-object)))
#<VLA-OBJECT IAcadDocument 00b94e14>

_$ (setq mspace (vla-get-modelspace acaddoc))
#<VLA-OBJECT IAcadModelSpace 01e42494>

_$ (setq PT1 (getpoint "\nSpecify First Point: "))
(228.279 430.843 0.0)

_$ (setq PT2 (getpoint "\nSpecify next point: " apt))
(503.866 538.358 0.0)

_$ (setq myline (vla-addline mspace (vlax-3d-point PT1)(vlax-3d-point PT2)))
#<VLA-OBJECT IAcadLine 00f9da94>

First we referenced Model Space, then we retrieved two points using the (getpoint) function.
We then used the (addline) method to draw our line, firstly converting the two point lists into variant arrays.

We can though, use Visual Lisp to obtain our two points. Try this :

_$ (setq util (vla-get-utility acaddoc))
#<VLA-OBJECT IAcadUtility 00f9e014>

_$ (setq PT1 (vla-getpoint util nil "\nSpecify First Point : "))
*#<variant 8197 ...>*

_$ (setq PT2 (vla-getpoint util PT1 "\nSpecify Second Point : "))
#<variant 8197 ...>

$ (setq myline (vla-addline mspace PT1 PT2))
#<VLA-OBJECT IAcadLine 00f9da94>

Because we used the Visual Lisp method to retrieve our points, the values were not returned as list, but as variant arrays. This means that we don't have to use (vlax-3d-point) to convert our points into variable arrays.
As you probably noticed from the line
     (setq util (vla-get-utility acaddoc))

the (vla-get) method, is a method of the "Utilitity" object.
Let's have a closer look.

```
Application
  (Object)
     |
     |
     |------------- Active Document
     |                (Object)
     |                   |
     |                   |
     |                   |
     |                   |------------- Utility
     |                   |                (Object)
     |                   |
```
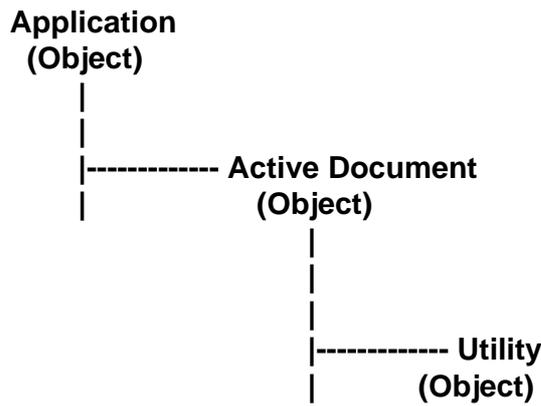
_$ (vl-load-com)

_$ (setq acaddoc (vla-get-activeDocument (vlax-get-acad-object)))
#<VLA-OBJECT IAcadDocument 00f5db9c>

_$ (setq util (vla-get-utility acaddoc))
#<VLA-OBJECT IAcadUtility 00f9e014>

_$ (vlax-dump-object util T)
; IAcadUtility: A series of methods provided for utility
purposes

; No properties

; Methods supported:
; AngleFromXAxis (2)
; AngleToReal (2)
; AngleToString (3)
; CreateTypedArray (3)
; DistanceToReal (2)
; GetAngle (2)
; GetCorner (2)
; GetDistance (2)
; GetEntity (3)
; GetInput ()
; GetInteger (1)
; GetKeyword (1)
; GetOrientation (2)
; GetPoint (2)
; GetReal (1)
; GetRemoteFile (3)
; GetString (2)
; GetSubEntity (5)
; InitializeUserInput (2)
; IsRemoteFile (2)
; IsURL (1)
; LaunchBrowserDialog (6)
; PolarPoint (3)
; Prompt (1)
; PutRemoteFile (2)
; RealToString (3)
; TranslateCoordinates (5)

Methods	Properties
AngleFromXAxis	Application
AngleToReal	
AngleToString	
CreateTypedArray	
DistanceToReal	
GetAngle	
GetCorner	
GetDistance	
GetEntity	
GetInput	
GetInteger	
GetKeyword	
GetOrientation	
GetPoint	
GetReal	
GetRemoteFile	
GetString	
GetSubEntity	
InitializeUserInput	
IsRemoteFile	
IsURL	
LaunchBrowserDialog	
PolarPoint	
Prompt	
PutRemoteFile	
RealToString	

T

Well, the Utility object doesn't seem to have any Properties, but it's got a treasure trove of Methods. We'll now have a wee look at a few of them.

First though, let's recap on the way we convert VBA methods to Visual Lisp methods. Let's look at the VBA "GetDistance" method :

GetDistance Method

Gets the distance from the prompt line or a selected set of points on the screen.

Syntax

RetVal = Object.GetDistance([Point][, Prompt])

- ■ **Object -Utility - The object or objects this method applies to.**
- ■ **Point - Variant (three-element array of doubles); input-only; optional**
  The 3D WCS coordinates specifying the base point. If this point is not provided, the user must input two points.
- ■ **Prompt - Variant (string); input-only; optional**
  The text to display to prompt the user for input.
- ■ **RetVal - Variant (double or array of doubles)**
  The distance from the prompt line or a selected set of points on the screen.

Remarks

AutoCAD pauses for user input of a linear distance and sets the return value to the value of the selected distance. The Point parameter specifies a base point in WCS coordinates. The Prompt parameter specifies a string that AutoCAD displays before it pauses. Both Point and Prompt are optional.

The AutoCAD user can specify the distance by entering a number in the current units format. The user can also set the distance by specifying two locations on the graphics screen. AutoCAD draws a rubber-band line from the first point to the current crosshair position to help the user visualize the distance. If the Point parameter is provided, AutoCAD uses this value as the first of the two points.

By default, GetDistance treats Point and the return value as three-dimensional points. A prior call to the InitializeUserInput method can force Point to be two-dimensional, ensuring that this method returns a planar distance.

Regardless of the method used to specify the distance or the current linear units (for example, feet and inches), this method always sets the return value to a double-precision floating-point value.

In Visual Lisp, the syntax would be :

RetVal = (vla-getdistance object [point] [prompt])

Example :

```
_$ (setq dist (vla-getdistance util nil "\nFirst Point : \n"))
535.428
```

Sometimes you retrieve a distance value as a string. Here's how you could convert it :

```
_$ (setq dist "123.45")
"123.45"
```

```
_$ (setq dist (vla-DistancetoReal util dist acDecimal))
123.45
```

Let's have a look at a few more "get" examples :

```
_$ (setq s (vla-getstring util 0 "Enter String: "))
"Kenny"
```

Note the "0" argument that disallows spaces.
Let's allow spaces this time :

```
_$ (setq s (vla-getstring util 1 "Enter String: "))
"Kenny Ramage is brilliant"
```

Let's get some Integers :

```
_$ (setq i (vla-getinteger util "\nEnter an Integer: "))
2
```

Now enter a real number for example 6.3

```
"Requires An Integer Value"
"Enter an Integer: "
```

Try it with a letter such as "A"
Same message.

Now we'll get some real's :

```
_$ (setq i (vla-getreal util "\nEnter and Integer: "))
5.7
```

Again, try it with a letter such as "C"

```
"Requires Numeric Value"
"Enter a Real: "
```

Now for an angle :

```
_$ (setq a (vla-getangle util nil "\nSelect Angle: "))
0.473349
```

Let 's give it a base point :

_$ (setq bpt (vla-getpoint util nil "/nSelect Base Point: "))
#<variant 8197 ...>

And use it to "rubber band" :

_$ (setq a (vla-getangle util bpt "\nSelect Angle: "))
0.707583

Now here is a couple of VERY interesting methods.

Want to select just one entity on the screen?

_$ (vla-getentity util 'obj 'ip "\nSelect Object: ")
nil
_$ obj
#<VLA-OBJECT IAcadLine 01ea57c4>
_$ ip
#<safearray...>

The reference to the Object is stored in the variable "obj" and the pickpoint is stored in variable "ip" in the form of a safearray.

Would you like to create a safearray very easily and quickly :

_$ (vla-createtypedarray util 'thearray vlax-vbDouble 1.2 2.3 0.0)
nil
_$ thearray
#<safearray...>

Your safearray is stored in the variable "thearray"

Let's now have a look at getting some keywords from the user.

The following example forces the user to enter a keyword by setting the first parameter of InitializeUserInput to 1, which disallows NULL input (pressing ENTER).

_$ (vla-InitializeUserInput util 1 "Line Arc Circle")
nil
_$ (setq kword (vla-GetKeyword util "Enter an Option (Line/Circle/Arc) : "))
"Line"

Let's have a closer look at "InitiliazeUserInput.

"InitiliazeUserInput" :

Syntax :

(vla-InitiliazeUserInput Object Bits Keyword)

Object : The object or objects this method applies to. In this case, the object is the Utility Object.

Bits : Integer; input-only
To set more than one condition at a time, add the values together in any combination. If this value is not included or is set to 0, none of the control conditions apply.

1	Disallows NULL input. This prevents the user from responding to the request by entering only [Return] or a space.
2	Disallows input of zero (0). This prevents the user from responding to the request by entering 0.
4	Disallows negative values. This prevents the user from responding to the request by entering a negative value.
8	Does not check drawing limits, even if the LIMCHECK system variable is on. This enables the user to enter a point outside the current drawing limits. This condition applies to the next user-input function even if the AutoCAD LIMCHECK system variable is currently set.
16	Not currently used.
32	Uses dashed lines when drawing rubber-band lines or boxes. This causes the rubber-band line or box that AutoCAD displays to be dashed instead of solid, for those methods that let the user specify a point by selecting a location on the graphics screen. (Some display drivers use a distinctive color instead of dashed lines.) If the POPUPS system variable is 0, AutoCAD ignores this bit.

64	Ignores Z coordinate of 3D points (GetDistance method only). This option ignores the Z coordinate of 3D points returned by the GetDistance method, so an application can ensure this function returns a 2D distance.
128	Allows arbitrary input—whatever the user types.

Keyword : Variant (array of strings); input-only; optional
The keywords that the following user-input method will recognize.

Remarks : Keywords must be defined with this method before the call to GetKeyword. Certain user-input methods can accept keyword values in addition to the values they normally return, provided that this method has been called to define the keyword. The user-input methods that can accept keywords are: GetKeyword, GetInteger, GetReal, GetDistance, GetAngle, GetOrientation, GetPoint, and GetCorner.

Let's have a look at another example :

A more user-friendly keyword prompt is one that provides a default value if the user presses ENTER (NULL input). Notice the minor modifications to the following example:

(vla-InitializeUserInput util 0 "Line Arc Circle")
nil
_$ (setq kword (vla-GetKeyword util "Enter an Option (Line/Circle/<Arc>) : "))
""
_$ (if (= kword "") (setq kword "Arc"))
"Arc"

Now we'll look at something very simple. Try this :

(vla-prompt util "\nPress any key to continue.....")
nil

And another interesting one :

_$ (setq pt1 (vla-getpoint util nil "\nFirst Point :"))
#<variant 8197 ...>

_$ (setq pt2 (vla-getpoint util pt1 "\nSecond Point :"))
#<variant 8197 ...>

_$ (setq theangle (vla-AngleFromXAxis util pt1 pt2))
0.459321

Of course, the result is returned in radians.

The Polar function, as you are probably well aware, is one of the most powerful function in the AutoLisp arsenal. Because of this, we are going to have a close look at the PolarPoint method. Lets have a look at

the syntax first :

"PolarPoint"

Syntax :

RetVal = (vla-PolarPoint Object Point Angle Distance)

Example : *(setq pt3 (vla-polarpoint pt2 0.5 20.0))*

- ■ **Object : Utility - The object or objects this method applies to.**
- ■ **Point : Variant (three-element array of doubles); input-only**
  The 3D WCS coordinates specifying the start point.
- ■ **Angle : Double; input-only**
  The angle in radians.
- ■ **Distance : Double; input-only**
  The distance in current units.
- ■ **RetVal : Variant (three-element array of doubles)**
  The 3D WCS coordinates at the specified angle and distance from a given point.

Here's a working example :
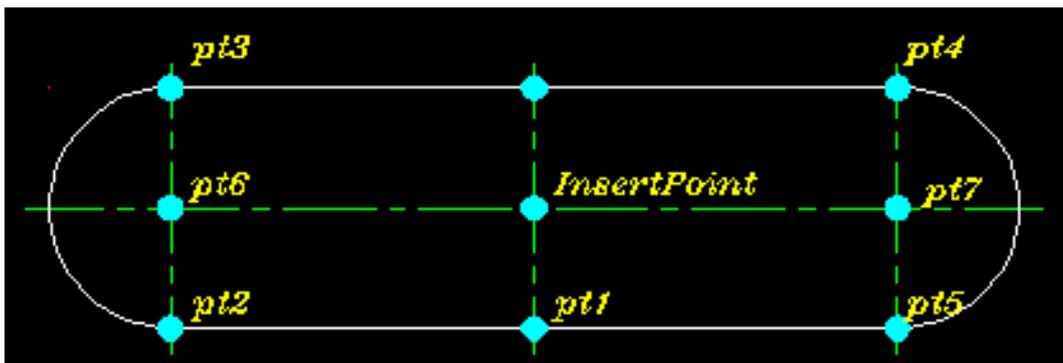
The following coding will draw slotted holes as per the above diagram.
It begins by asking the user for an insertion point, the slot length, and then the slot diameter using Visual Lisp Get methods.
It then uses the PolarPoint method to calculate the other points required to draw the slot. Then, using the AddLine and AddArc methods, it draws the slotted hole.

```
 ;CODING STARTS HERE
(defun c:slot ( / acaddoc util mspace ip lg
                 dia P1 P2 P3 P4 P5 P6 P7)

;load VL functions
(vl-load-com)

;obtain reference to the Active Document
(setq acaddoc (vla-get-activeDocument (vlax-get-acad-object)))

;obtain reference to Utilities
(setq util (vla-get-utility acaddoc))
```

```lisp
;obtain reference to Model Space
(setq mspace (vla-get-modelSpace acaddoc))

;get the insertion point
(setq ip (vla-getpoint util nil "\nInsertion Point: "))

;get the length
(setq lg (vla-getreal util "\nEnter Slot Length: "))

;get the diameter
(setq dia (vla-getreal util "\nEnter Slot Diameter: "))

;calculate all the points
(setq P1 (vla-polarpoint util IP (dtr 270.0) (/ dia 2)))
(setq P2 (vla-polarpoint util P1 (dtr 180.0) (/ lg 2)))
(setq P3 (vla-polarpoint util P2 (dtr 90.0) dia))
(setq P4 (vla-polarpoint util P3 0 lg))
(setq P5 (vla-polarpoint util P4 (dtr 270.0) dia))
(setq P6 (vla-polarpoint util P2 (dtr 90.0) (/ dia 2)))
(setq P7 (vla-polarpoint util P5 (dtr 90.0) (/ dia 2)))

;draw the lines
(vla-AddLine mspace p2 p5)
(vla-AddLine mspace p3 p4)

;add the arcs
(vla-AddArc mspace p6 (/ dia 2) (dtr 90.0) (dtr 270.0))
(vla-AddArc mspace p7 (/ dia 2) (dtr 270.0) (dtr 90.0))

   (princ)

);defun
----------

;convert degrees to radians

(defun DTR (a)

 (* pi (/ a 180))

);defun

----------

(princ)

;CODING ENDS HERE
```

Visual Lisp and VBA

In this tutorial we are going to do something a bit different. We're going to look at how we can use Visual Basic for Applications within our Visual Lisp code, especially in regards to dialog boxes. Firstly though, let's have a look at how we are going to go about loading and running the VBA application that we want to use. Copy and paste this into Notepad and save it as "Al-Eval.lsp" :

```
(defun c:al-eval ()

(vl-load-com)

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

(vla-eval applic (strcat "MsgBox \"Hello Everybody\"" ", "
 "vbInformation" ", " "\"AfraLisp\""))

(vla-eval applic (strcat "MsgBox \"Are You Fine?\"" ", "
"vbQuestion" ", " "\"CAD Encoding\""))

(vla-eval applic (strcat "MsgBox \"Or, are you not Well?\"" ", "
"vbCritical" ", " "\"VBA Expresso\""))

(alert "\nAnd this is the boring\nAutoCAD message box!!")

(princ)

);defun
```

Now load and run the routine. Four dialog boxes should appear, one after the other.
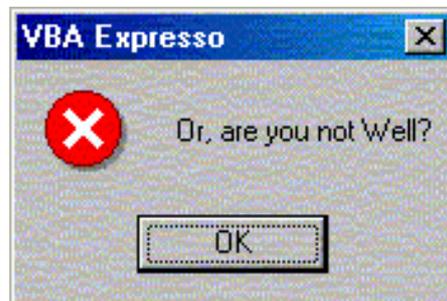

The first three dialogs, similar to the above, are standard VBA message boxes, and the fourth the AutoLisp alert box. By using the Visual Lisp "eval" method, we were able to "call" a standard VBA function from within Visual Lisp.

Let's get a bit more clever now, and try and pass information backwards and forwards between Visual Lisp and VBA.

Open the VBA Editor and create a new Project. Insert a UserForm and add a TextBox and two command buttons, retaining their default names. Your UserForm should look something like this :
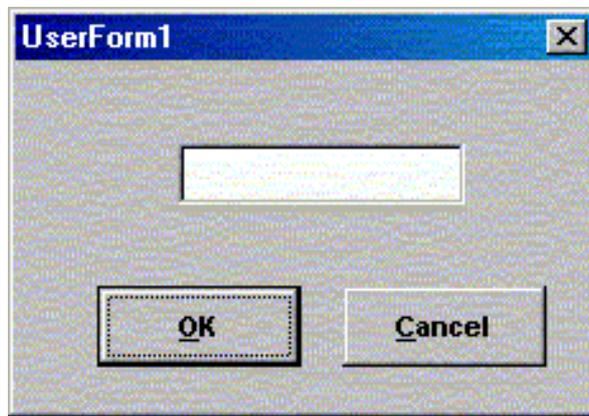
Now add the following coding to the the Click events for the OK and Cancel buttons :

```
  Private Sub CommandButton1_Click()

ThisDrawing.SetVariable "USERS1", TextBox1.Text
End

End Sub
'--------------------------------------------------------
Private Sub CommandButton2_Click()
End
End Sub
```

Save you Project as "Testdial.dvb".

Now open Notepad and copy and paste the following :

```
(defun c:testdial ()

(vl-load-com)

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

(if (findfile "testdial.dvb")

   (progn

      (vl-vbaload "testdial.dvb")

      (vla-eval applic "userform1.show")

      (alert (strcat "You entered : " (getvar "users1")))

   );progn

      (alert "\nCannot find DVB file")

  );if

(princ)

);defun
```

```
(princ)
```

Load and run "Testdial.lsp". A dialog like this should appear :
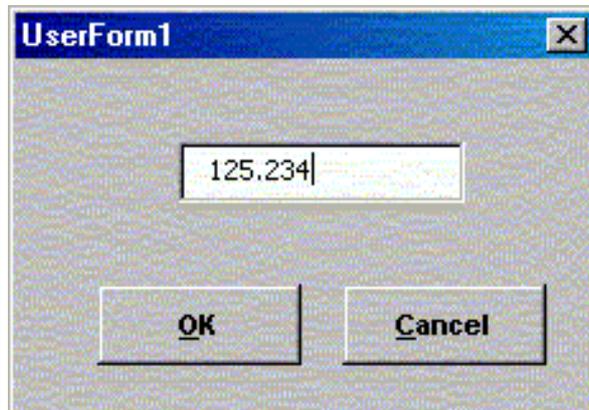
Enter something into the TextBox and select "OK".
An alert box like this should appear:
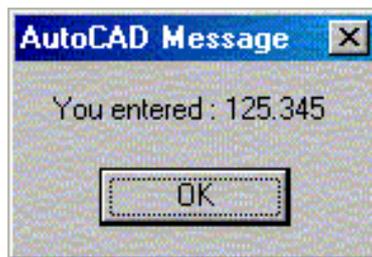
This is what happened :

- **First, we loaded and called the VBA dialog.**
- **Secondly, we entered the information.**
- **Still in VBA, we stored the information into a User system variable.**
- **We then closed the dialog.**
- **Back in Visual Lisp, we retrieved the information from the User system variable and displayed it in an AutoLisp alert box.**

Next, we'll have a look at a more practical use for this procedure.

In this next exercise, we're going to design a drawing tile block application. The program will be written in Visual Lisp, but we will call a VBA program and dialog to display and return our title block data.

OK, open up your VBA Editor and start a new Project. Insert a UserForm and add 5 TextBoxes, 5 Labels and two command buttons. Retain their default names. Your dialog should look like this :
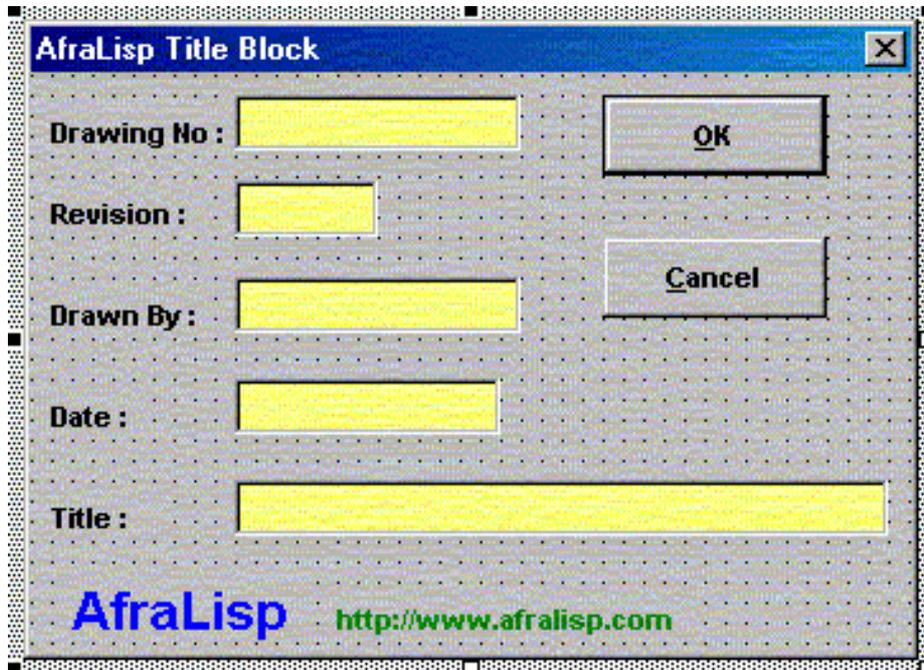
Now enter the following coding :

```
Option Explicit
'---------------------------------------------------
Private Sub CommandButton1_Click()

'set the user system variable
ThisDrawing.SetVariable "UserI1", 1

'retrieve the text box values and store
'them in the user system variables
ThisDrawing.SetVariable "UserS1", TextBox1.Text
ThisDrawing.SetVariable "UserS2", TextBox2.Text
ThisDrawing.SetVariable "UserS3", TextBox3.Text
ThisDrawing.SetVariable "UserS4", TextBox4.Text
ThisDrawing.SetVariable "UserS5", TextBox5.Text
End

End Sub

'---------------------------------------------------
Private Sub CommandButton2_Click()

'set the user system variable
ThisDrawing.SetVariable "UserI1", 0
End

End Sub

'---------------------------------------------------
Private Sub UserForm_Initialize()
```

```
'retrieve user system variables and populate
'the text boxes
TextBox1.Text = ThisDrawing.GetVariable("UserS1")
TextBox2.Text = ThisDrawing.GetVariable("UserS2")
TextBox3.Text = ThisDrawing.GetVariable("UserS3")
TextBox4.Text = ThisDrawing.GetVariable("UserS4")
TextBox5.Text = ThisDrawing.GetVariable("UserS5")

'set the focus to the first text box
TextBox1.SetFocus
TextBox1.SelStart = 0
TextBox1.SelLength = Len(UserForm1.TextBox1.Text)

End Sub
```

Save this Project as "AL-VBA.DVB".

Now copy and paste this into Notepad and save it as "VL-VBA.LSP" :

```
;CODING STARTS HERE
;
;All Tutorials and Code are provided "as-is" for purposes of instruction and
;utility and may be used by anyone for any purpose entirely at their own risk.
;Please respect the intellectual rights of others.
;All material provided here is unsupported and without warranty of any kind.
;No responsibility will be taken for any direct or indirect consequences
;resulting from or associated with the use of these Tutorials or Code.
;****************************************************************************
;                    AfraLisp
;            http://www.afralisp.com
;             afralisp@afralisp.com
;             afralisp@mweb.com.na
;****************************************************************************
;This application will extract attributes from a block and display them in a
;VBA dialog box. The attributes will then be updated.
;
;Dependencies : VL-VBA.DVB and VL-VBA.DWG are
;required and must be within the AutoCAD search path.
;
;Usage : Open VL-VBA.DWG then load and run VL-VBA.LSP.
;****************************************************************************
;

(prompt "\nVL-VBA Loaded....Type VL-VBA to run.....")

(defun C:VL-VBA ( / thisdrawing applic ssets newsset filter_code filter_value
                   item theatts attlist theattribute1 theattribute2 theattribute3
                   theattribute4 theattribute5)

;load visual lisp extensions
(vl-load-com)

;retrieve reference to the active document
(setq thisdrawing (vla-get-activedocument (vlax-get-acad-object)))
```

```lisp
;retrieve reference to the selection set collection
(setq ssets (vla-get-selectionsets thisdrawing))

;check if the selection set exists - $Set
(if (vl-catch-all-error-p (vl-catch-all-apply 'vla-item (list ssets "$Set")))

  ;if it doesn't create a new one
  (setq newsset (vla-add ssets "$Set"))

    ;if it does exist
    (progn

      ;delete it
      (vla-delete (vla-item ssets "$Set"))

      ;then create a new one
      (setq newsset (vla-add ssets "$Set"))

    );progn

);if

;create a single element array - integer
(setq filter_code (vlax-make-safearray vlax-vbinteger '(0 . 0)))

;create a single element array - variant
(setq filter_value (vlax-make-safearray vlax-vbvariant '(0 . 0)))

;filter for name - code 2
(vlax-safearray-fill filter_code '(2))

;filter for block name - attab-info
(vlax-safearray-fill filter_value '("attab-info"))

;filter the drawing for the block
(vla-select newsset acSelectionSetAll nil nil filter_code filter_value)

  ;if the block is found
  (if (>= (vla-get-count newsset) 1)

  ;display the dialog
  (ddisplay)

  ;if the block is not found
  (alert
    "\nIncorrect Drawing Sheet
   \n    Use Manual Edit"
  )

 );if

 ;release all objects
 (vlax-release-object thisdrawing)
 (vlax-release-object applic)

;finish clean
(princ)

);defun
```

```
;...****************************************************
;;;

(defun ddisplay (/)

;retrieve the block reference
(setq item (vla-item newsset 0))

;retrieve the attributes
(setq theatts (vla-getattributes item))

;convert to a list
(setq attlist (vlax-safearray->list (variant-value theatts)))

;extract the attributes
(mapcar 'set '(theattribute1 theattribute2 theattribute3
                 theattribute4 theattribute5) attlist)

 ;extract the text strings from the attributes
 ;then put the strings into system variables
 (vla-SetVariable thisdrawing "USERS1" (vla-get-textstring theattribute1))
 (vla-SetVariable thisdrawing "USERS2" (vla-get-textstring theattribute2))
 (vla-SetVariable thisdrawing "USERS3" (vla-get-textstring theattribute3))
 (vla-SetVariable thisdrawing "USERS4" (vla-get-textstring theattribute4))
 (vla-SetVariable thisdrawing "USERS5" (vla-get-textstring theattribute5))

;find the VBA file
(if (findfile "vl-vba.dvb")

;if it's found, do the following
(progn

;load the VBA file
(vl-vbaload "vl-vba.dvb")

;get a reference to the application object
(setq applic (vlax-get-acad-object))

;display the VBA dialog
(vla-eval applic "userform1.show")

 ;if OK was selected
 (if (= (getvar "USERI1") 1)

 ;do the following
 (progn

  ;update the attribute textstrings
  (vla-put-textstring theattribute1 (vla-GetVariable thisdrawing "USERS1"))
  (vla-put-textstring theattribute2 (vla-GetVariable thisdrawing "USERS2"))
  (vla-put-textstring theattribute3 (vla-GetVariable thisdrawing "USERS3"))
  (vla-put-textstring theattribute4 (vla-GetVariable thisdrawing "USERS4"))
  (vla-put-textstring theattribute5 (vla-GetVariable thisdrawing "USERS5"))

  ;update the block
  (vla-update newsset)

  ;regen the drawing
  (vl-cmdf "REGEN")
```

```
    );progn

    );if

);progn

;couldn't find the VBA file
(alert "\nCannot find VBA file.")

);if

);defun

;...*******************************************************
;;;

;load clean
(princ)

;...*******************************************************
;;;
;
;CODING ENDS HERE
```

Now, open the the drawing "AL-VBA.DWG".
A title block should appear :

Load and run "VL-VBA.LSP". Your dialog should appear in all it's glory containing the attribute data from the title block.

Change a few of the title block values and then press "OK". Your title block drawing should be updated to reflect your new values. Good one hey?

Visual Lisp and HTML

Time for another wee bit of a break for me. By sleuth of hand and the use of threats to various appendages stuck on his body, I finally conned someone else into writing this section for me. Say hi to David Stein everyone, ("*Hi Dave!!*")
If you don't know Dave, well he is the gentleman responsible for "DSX Tools" and the "Visual Lisp Bible." You can find his new permanent home at http://www.dsxcad.com. Pop along as it's well worth the visit. (just check for your wife and wallet when you leave.)

Okey dokey. Dave is the author of this nice little program written using Visual Lisp that will not only write the Layer Names to an HTML file, but will also write the Layer properties and status. Let's have a look at the coding.

Copy and paste this into Notepad and save it as "DumpLayers.lsp."

```
;; 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)

;load the VL extensions
(vl-load-com)

;get reference to AutoCAD
(setq acad (vlax-get-acad-object)

      ;reference the drawing
      doc (vla-get-activedocument acad)

      ;get the drawing name
      dwg (vla-get-name doc)

      ;get the drawing path
      path (vla-get-path doc)

    ;get the layers cpllection
    layers (vla-get-layers doc)

);setq

;process each layer
(vlax-for each layers

   ;get the layer name
   (setq name (vla-get-name each)

         ;get the layer color
          col (itoa (vla-get-color each))

         ;get the linetype
         ltp (vla-get-linetype each)
```

```
            ;get the lineweight
            lwt (itoa (vla-get-lineweight each))

            ;get the plotstyle
            pst (vla-get-plotstylename each)

            ;on-off status
            onoff (if (= :vlax-true (vla-get-layeron each))
             "ON" "OFF")

            ;frozen-thawed status
            frz (if (= :vlax-true (vla-get-freeze each))
            "FROZEN" "THAWED")

            ;list them
            dat (list name col ltp lwt pst onoff frz)

            ;add to main list
            olist (cons dat olist)

    );setq

); vlax-for

;release all objects
(vlax-release-object layers)
(vlax-release-object doc)
(vlax-release-object acad)

;create the HTML file
(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)

   ;inform the user
   (princ "\nReport finished! Opening in browser...")

   ;open the HTML report in the browser
   (vl-cmdf "_.browser" outfile)
```

```
        )

      ( T (princ "\nUnable to open output file.") )
      )
    )
    ( T (princ "\nUnable to get layer table information.") )
    )
);defun

(princ)
```

Load and run this routine. The HTML file should automatically open in your browser.
Hey, is that not nice? All your layers, layer properties and status all nicely tabulated in a report!

Acknowledgments and Links

 A big thanks to :

My wife Heather.
My Mum and Dad
EVERYONE at VBA Expresso.
The lads from BedRock - Pete, Eddie and Mike.
Frank Zander for his generosity and support.

And a BIG thank you to Marie and Jessie Rath for at least trying to control that reprobate Randall. (Thanks for everything pal.)

If I've forgotten someone, hey life's tough.........

Some of the best Links on the Web :

AfraLisp - http://www.afralisp.com

VBA Expresso - http://www.vbdesign.net/cgi-bin/ikonboard.cgi

CAD Encoding - The Journal - http://www.cadencoding.com

VB Design - http://www.vbdesign.net

Contract CADD Group - http://www.contractcaddgroup.com

CopyPaste Code - http://www.copypastecode.com

DsxCad - http://www.dsxcad.com

QnA~Tnt - http://www.dczaland.com/appinatt

BedRock - http://www.bedrockband.com