SD196926

Using the Visual LISP Extension with AutoLISP

Lee Ambrosius
Autodesk, Inc.

Learning Objectives

- Learn how to access and store values in the Windows registry
- Learn how to register application, document, and other types of reactors
- Learn how to open drawing files outside of the AutoCAD application
- Learn how to manipulate the Windows environment and MS Office applications with AutoLISP

Description

The AutoLISP programming language is very powerful by itself, but so much more can be done with AutoLISP by using the Visual LISP extension. The Visual LISP extension lets you access parts of the Windows operating system and the AutoCAD application that AutoLISP can't normally access. In this session, you will learn how to access the Windows registry, work with events and reactors, utilize the AutoCAD and Windows ActiveX libraries, and connect to other Windows applications like Microsoft Word or Microsoft Excel. This session is not recommended for those new to AutoLISP.

Speaker(s)

Lee Ambrosius is a Principal Learning Experience Designer at Autodesk, Inc., for the AutoCAD and AutoCAD LT products on Windows and Mac OS. He works primarily on the customization, developer, and CAD administration documentation along with the user documentation. Lee has presented at Autodesk University for almost 15 years on a wide range of topics, from general AutoCAD customization to programming with the ObjectARX technology. He has authored several AutoCAD-related books, with his most recent project being *AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond*. When Lee isn't writing, you can find him running or cycling and engaging the AutoCAD community via various platforms (forums, AutoCAD related blogs, and Twitter).

Twitter: @leeambrosius
Email: lee.ambrosius@autodesk.com
Blog: http://hyperpics.blogs.com

1 What is the Visual LISP Extension?

AutoCAD 2000 marked a major expansion of the AutoLISP language and platform known as *Visual LISP*. There were several major changes introduced with Visual LISP:

- Dedicated development environment known as the Visual LISP IDE which can be displayed using the VLIDE command
- Hundreds of new functions were introduced that improved access to the objects stored in a drawing
- Support for COM (Component Object Model) libraries exposed by other applications installed on the same system

Note: COM libraries are part of Microsoft's ActiveX technology which results in the terms COM and ActiveX to be used interchangeably; so, if you hear ActiveX the person is most likely talking about COM

Using the AutoCAD COM library, you can access drawing objects in a much more direct approach than having to remember DXF codes or rely on the use of commands. Prior to using any of the new functions available with Visual LISP, you must call the `vl-load-com` function to ensure the extension has been loaded. If the extension is already loaded, there is no performance impact in calling the function multiple times.

Create a Circle

Below are several different code examples that show the creation of a circle and changing it to the ACI color of 5 (blue). The first three examples utilize AutoLISP functions available in AutoCAD 2012 and later, while the last example utilizes the functions available through the Visual LISP and AutoCAD COM library.

```
;; Create a blue circle using commands
(defun c:CreateCircle-Cmds ( / )
  ; Create circle
    (command "._circle" '(5 5 0) 1.75)

  ; Change the color of the circle
    (command "._change" (entlast) "" "_p" "_c" "5" "")
 (princ)
)
```

```
;; Create a blue circle without using commands
(defun c:CreateCircle-Dxf ( / circObj edList)
  ; Create circle
    (setq circObj (entmakex '((0 . "CIRCLE")   ; Entity type
                              (10 5.0 5.0 0.0) ; Center point
                              (40 . 1.75)      ; Radius
                             ))
    )

  ; Get the entity data for the circle object
    (setq edList (entget circObj))

  ; Change the color of the circle
    (if (assoc 62 edList)
      (setq edList (subst (cons 62 5) (assoc 62 edList) edList))
      (setq edList (append edList (list (cons 62 5))))
    )

  ; Update the object
    (entmod edList)
    (entupd circObj)
 (princ)
)

;; Create a blue circle without commands and DXF manipulation
(defun c:CreateCircle-SetProp ( / circObj)
  ; Create circle
    (setq circObj (entmakex '((0 . "CIRCLE")   ; Entity type
                              (10 5.0 5.0 0.0) ; Center point
                              (40 . 1.75)      ; Radius
                             ))
    )

  ; Change the color of the circle
    (setpropertyvalue (entlast) "Color" "5")
 (princ)
)
```

```
;; Create a blue circle using the Visual LISP extension
(defun c:CreateCircle-VL ( / acadObj docObj spaceObj cenPoint circObj)
  ; Create the circle
    ; Get the AutoCAD application object
    (setq acadObj (vlax-get-acad-object))

    ; Get the current drawing
    (setq docObj (vla-get-activedocument acadObj))

    ; Get the current space (model or layout)
    (if (= (vla-get-activespace docObj) acModelSpace)
      (setq spaceObj (vla-get-modelspace docObj))
      (setq spaceObj (vla-get-paperspace docObj))
    )

    ; Create a 3 element array that will represent a coordinate value
    (setq cenPoint (vlax-make-safearray vlax-vbdouble '(0 . 2)))
    (vlax-safearray-fill cenPoint '(5.0 5.0 0.0))

    ; Add the circle object to the drawing
    (setq circObj (vla-addcircle spaceObj cenPoint 1.75))

  ; Change the color of the circle
    (vla-put-color circObj 5)
 (princ)
)
```

Modifying Objects and Setting Some Properties to ByLayer

Below are several different code examples that step through the objects in a drawing and sets several properties to ByLayer. The properties that are manipulated by the examples are color, linetype, and lineweight. The first three examples utilize AutoLISP functions available in AutoCAD 2012 and later, while the last example utilizes the functions available through the Visual LISP and AutoCAD COM library.

```
;; Change all objects to ByLayer using commands
(defun c:ChangeAll2ByLayer-Cmds ( / )
    (command "._-setbylayer" "_s" "_c" "_lt" "_lw" "" "_all" ""
             "_y" "_y")
 (princ)
)
```

```
;; Change all objects to ByLayer without using commands
(defun c:ChangeAll2ByLayer-Dxf ( / ent edList)
  ; Get the first object in the drawing
    (setq ent (entnext))

  ; Step through each object in the drawing
    (while ent
  ; Get the entity data for the object
      (setq edList (entget ent))

      ; Remove assigned color
      (if (assoc 62 edList)
         (setq edList (subst (cons 62 256) (assoc 62 edList) edList))
      )
      (if (assoc 420 edList)
        (setq edList (vl-remove (assoc 420 edList) edList))
      )
      (if (assoc 430 edList)
        (setq edList (vl-remove (assoc 430 edList) edList))
      )

      ; Remove assigned linetype
      (if (assoc 6 edList)
         (setq edList (subst (cons 6 "BYLAYER")
                             (assoc 6 edList) edList))
      )

      ; Remove assigned lineweight
      (if (assoc 370 edList)
         (setq edList (subst (cons 370 -1) (assoc 370 edList) edList))
      )

      ; Update the object
      (entmod edList)
      (entupd ent)

  ; Get the next object in the drawing
      (setq ent (entnext ent))
    )
 (princ)
)
```

```
;; Change all objects to ByLayer without commands and DXF manipulation
(defun c:ChangeAll2ByLayer-SetProp ( / ent)
  ; Get the first object in the drawing
    (setq ent (entnext))

  ; Step through each object in the drawing
    (while ent
      ; Remove assigned color
      (setpropertyvalue ent "Color" "256")
      ; Remove assigned linetype
      (setpropertyvalue ent "LinetypeId"
         (getpropertyvalue (tblobjname "ltype" "BYLAYER") "ObjectId"))
      ; Remove assigned lineweight
      (setpropertyvalue ent "LineWeight" -1)

  ; Get the next object in the drawing
      (setq ent (entnext ent))
    )
 (princ)
)

;; Change all objects to ByLayer using the Visual LISP extension
(defun c:ChangeAll2ByLayer-VL ( / acadObj docObj spaceObj for-item)
    ; Get the AutoCAD application object
    (setq acadObj (vlax-get-acad-object))

    ; Get the current drawing
    (setq docObj (vla-get-activedocument acadObj))

    ; Get the current space (model or layout)
    (if (= (vla-get-activespace docObj) acModelSpace)
      (setq spaceObj (vla-get-modelspace docObj))
      (setq spaceObj (vla-get-paperspace docObj))
    )

    ; Step through the current space
    (vlax-for for-item spaceObj
      ; Remove assigned color
      (vla-put-color for-item acByLayer)
      ; Remove assigned linetype
      (vla-put-linetype for-item "BYLAYER")
      ; Remove assigned lineweight
      (vla-put-lineweight for-item acLnWtByLayer)
    )
 (princ)
)
```

Accessing the Top Level of the AutoCAD COM Library

Now that you have seen a few examples that utilize the Visual LISP extension, it's time for a basic understanding of the AutoCAD COM library. The AutoCAD COM library is a hierarchical

structure with the main object being the AutoCAD Application object at the top. Once you have a reference to the AutoCAD Application object, you can work your way down the hierarchy:

- Active Document or Documents Collection
    - Named Tables/Dictionaries (Blocks, text styles, dimension styles, …)
        - Table Records and Dictionary Entries

As an example, if you want to access a circle in Model space of a drawing you would need to do the following at a high level:

1. Get the AutoCAD Application object
2. Get the Active Document (aka *drawing*) or another open document
3. Get the Model space of the drawing
4. Get the circle in the Model space collection

The AutoCAD Application object is obtained with the `(vlax-get-acad-object)` function, it doesn't take any argument values.

```
(vlax-get-acad-object)
#<VLA-OBJECT IAcadApplication 00007ff7f2b1b0a8>
```

Identifying the Properties and Methods Available for an Object

After you have an ActiveX object (`vla-object`) like the AutoCAD Application object, you can list the object's available properties and methods with the `vlax-dump-object` function a flag value that evaluates to true, such as `T`. The syntax for the `vlax-dump-object` function is

```
(vlax-dump-object obj [flag])
```

`obj` is an object of the `vla-object` data type, such as the AutoCAD Application object returned by the `(vlax-get-acad-object)` function.

For example, the `(vlax-dump-object (vlax-get-acad-object) T)` statement returns a list of properties and methods available for the AutoCAD application object. Here is part of the output returned by the `vlax-dump-object` function:

```
; IAcadApplication: An instance of the AutoCAD application
; Property values:
;   ActiveDocument = #<VLA-OBJECT IAcadDocument 0000023b4d814a78>
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00007ff7f2b1b0a8>
…
;   Name (RO) = "AutoCAD"
;   Path (RO) = "C:\\Program Files\\Autodesk\\AutoCAD 2019"
…
; Methods supported:
;   Eval (1)
…
;   Quit ()
;   ZoomAll ()
…
```

Once you know which properties and methods an object supports, a prefix can be appended to the name of the property or method to identify the function that can be used in AutoLISP with the Visual LISP extension. To access the properties of an object, you either prefix the property name with `vla-get-` or `vla-put-`; `vla-get-` returns the current value of a property while `vla-put-` assigns a new value to a property.

For example, to get the location in which the AutoCAD executable is installed, you would query the Path property of the AutoCAD Application object. Since you want to get the current value of the Path property, you would use the function `vla-get-path`.

```
(vla-get-path (vlax-get-acad-object))
"C:\\Program Files\\Autodesk\\AutoCAD 2019"
```

- `vla-get-` functions expect one argument which is the object to query
- `vla-put-` functions expect two values which are the object to update and the value to be assigned to the object being updated

Note: Property names with (RO) after them are read-only and their currently assigned values can't be changed using the function prefixed with `vla-put-`.

To use a method, you prefix the name of the method with `vla-`. All methods require at least one argument value, and that is the object in which the method should affect. For example, to display all visible objects in a drawing you can use the `vla-zoomall` function.

```
(vla-zoomall (vlax-get-acad-object))
```

If you look at the output from the `vlax-dump-object` function, it indicates the number of additional arguments that you will need to provide besides the required argument value. For example, `Eval (1)` from the output indicates the `vla-eval` function expects a reference to the AutoCAD Application object along with a string that represents the VBA expression to evaluate.

Tip: You can use the ActiveX Reference Guide in the Developer Documentation section of the AutoCAD Online Help system to identify the return and expected argument values of the properties and methods in the AutoCAD COM library.

Passing Data Between AutoCAD COM and Legacy AutoLISP Functions

There will be times when you might need to update existing programs to leverage functionality found in the AutoCAD COM and other third-party libraries with the AutoLISP programming language. Entity names (`ename`), which are returned by functions such as `entnext` or `entsel`, returned by legacy AutoLISP functions must be transformed to the `vla-object` data type in order for them to be used with methods from the AutoCAD COM library, and the same holds true if you want to work with a `vla-object` with some of the legacy AutoLISP functions that expect an `ename` data type. The `type` function can be helpful in determining the data type of a returned value or one that has been assigned to a user-defined variable.

```
(type (vlax-get-acad-object))
VLA-OBJECT

(type (car (entsel)))
ENAME
```

The following lists the two functions that are used to convert an `ename` to a `vla-object` or a `vla-object` to an `ename`:

- `vlax-ename->vla-object` – Transforms an entity name (`ename`) into a `vla-object`
- `vlax-vla-object->ename` – Transforms a `vla-object` into an entity name (`ename`)

Along with transforming `ename` and `vla-object` data types, you might need to convert lists to safearrays and back. The methods of the AutoCAD COM library don't understand what a list is,

but they do understand safearrays which are like each other in functionality like the list data type in that they can hold multiple indexed values.

You can make a safearray using the `vlax-make-safearray` function and then populate it using a few different functions, but if you are converting a safearray to a list you most likely will use the `vlax-safearray-fill` to copy the values in a list to an array. For example, if you wanted to convert a list that represents a coordinate to a safearray you would do something like the following:

```
(setq point (vlax-make-safearray vlax-vbDouble '(0 . 2)))
#<safearray...>

(vlax-safearray-fill point '(5.0 5.0 0.0))
#<safearray...>
```

Note: One thing to be aware of when it comes to safearrays is that they are often declared as containing values of a specific type rather than lists that could contain any supported data type. The `vlax-safearray-type` function can be used to identify the data type in which the safearray was declared.

If a function returns a safearray and you need a list, you can use the `vlax-safearray->list` function to convert a safearray to a list that can then be used with legacy AutoLISP functions. The following shows how to convert the safearray assigned to the user-defined variable *`point`* to a list:

```
(vlax-safearray->list point)
(5.0 5.0 0.0)
```

2 Store and Access Information Later

AutoCAD commands often store previously used values, allowing them to quickly access those values the next time they are used in the same drawing session, between drawing sessions, or globally across all drawings and sessions. The way information is stored for use later by a command depends on the task in which a command performs.

The following are several different examples of how AutoCAD stores values for different commands:

- **CIRCLE command** – The last radius entered is maintained within only the current drawing session, and once the drawing is closed the value is lost.
- **FILLET command** – The last fillet radius entered is maintained between sessions, so if the drawing is closed and re-opened the last value entered is the one that is used by the FILLET command the next time it is started.
- **OPTIONS command** – The last profile set current is the profile that is used for each drawing that is opened and when AutoCAD is closed and re-opened.

When creating custom applications, often persisting information is just done within the current drawing and session, so if the drawing is closed the routine usually uses a set of default values the next time it is loaded. While this might be ideal for most custom routines, it might not provide the best user experience in all situations.

Visual LISP allows you to persist values for use

- In the current drawing only while it remains open with the `setq` function
- In all open drawings via the blackboard feature
- In all open drawings and across multiple sessions with the Windows Registry
- In a drawing across multiple sessions with Xdata and dictionaries

Accessing Variables across All Drawings with the Blackboard

Variables are used to store values for use later and typically are defined with the `setq` function. While not all variables need to be accessed from outside of the current drawing, there are times when you might want to access a value across all open drawings in the current session. The Visual LISP extension provides a feature known as the *blockboard*.

The blackboard allows you to create user-defined variables at the application level instead of the current drawing only. For example, you could use the blackboard to access the name of the most recent file used by your custom program instead of storing the value in the Windows Registry (Windows) or Plist file (Mac OS X).

User-defined variables can be created and assigned a value on the blackboard with the `vl-bb-set` function, while the current value of a user-defined value on the blackboard can be retrieved using the `vl-bb-ref` function.

The syntax for the `vl-bb-set` and `vl-bb-ref` functions is:

```
(vl-bb-set 'variable value)
(vl-bb-ref 'variable)
```

The following demonstrates storing and accessing values with the `setq`, `eval`, `vl-bb-ref`, and `vl-bb-set` functions:

```
; Returns the value of notes-template-file in the current drawing
(eval notes-template-file)
nil

; Defines a variable named notes-template-file in the current drawing
(setq notes-template-file "notes2.xml")
"notes.xml"

; Returns the value of notes-template-file on the blackboard
(vl-bb-ref 'notes-template-file)
nil

; Defines a variable named notes-template-file on the blackboard
(vl-bb-set 'notes-template-file "notes.xml")
"notes2.xml"

; Returns the value of notes-template-file created with setq
(eval notes-template-file)
"notes.xml"

; Gets the value of the variable notes-template-file on the blackboard
(vl-bb-ref 'notes-template-file)
"notes2.xml"
```

After trying the above AutoLISP statements in a drawing, create a new drawing and try to access the user-defined variable again with the `eval` and `vl-bb-ref` functions. You should get the following results:

```
(eval notes-template-file)
nil

(vl-bb-ref 'notes-template-file)
"notes.xml"
```

Windows Registry

Rather than only being able to access values while a drawing or application remains open, it can be beneficial to persist data across multiple sessions of AutoCAD. Visual LISP allows you to access the Windows Registry (or its equivalent a Plist file on Mac OS) to persist data across multiple sessions of AutoCAD. The Windows Registry can be great for storing paths to custom blocks or the discipline in which a user self identifies.

The Windows Registry can be modified and viewed directly using the Registry Editor which can be started using the Run command from the Windows Start menu and typing ***regedit.exe***.

Important: When working with the Windows Registry, exercise caution as you don't want to overwrite or delete a key that you didn't create or don't understand what it's intended purpose is. If you start altering values and keys for an application other than your own, the application may become unstable the next time it is used.

Visual LISP offers four functions that are used to read and write values to and from the Windows Registry. The functions that are available for working with the Windows Registry are listed in the following table.

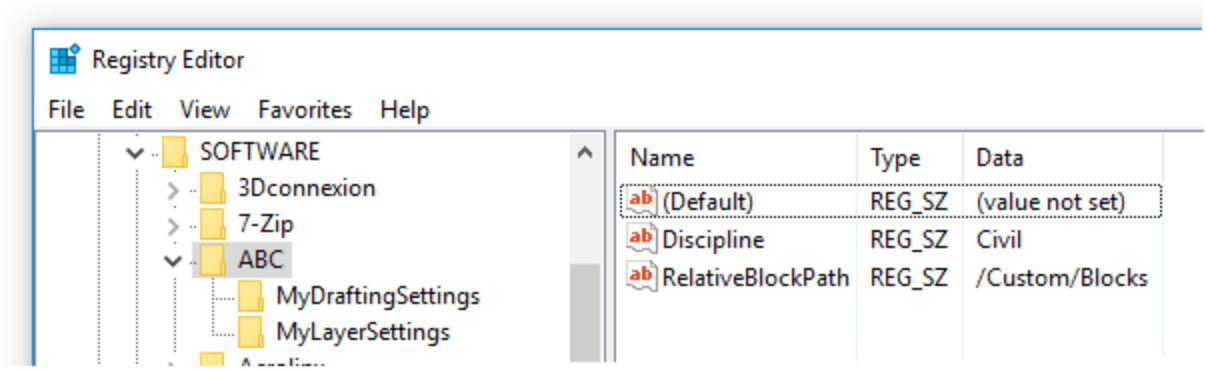
Function with syntax	Description
(vl-registry-delete key [value])	Removes a key or value from the registry
(vl-registry-descendents key [value(s)])	Returns a listing of either subkeys or value names of the key specified
(vl-registry-read key [value])	Reads the data from the default value of a key or a specific value
(vl-registry-write key [value data])	Writes a single piece of data from to the default value of a key or a specific value

```
;; Command demonstrates the use of the functions
;; used to work with the Windows Registry.
(defun c:WinReg ( / reg-val)
  (alert
    (strcat "Current Discipline: "
      (if (setq reg-val
            (vl-registry-read
                 "HKEY_CURRENT_USER\\Software\\ABC" "Discipline"))
                 reg-val "<Missing>")
      )
  )

  (vl-registry-write "HKEY_CURRENT_USER\\Software\\ABC"
                     "Discipline" "Civil")
  (vl-registry-write "HKEY_CURRENT_USER\\Software\\ABC"
                     "RelativeBlockPath" "/Custom/Blocks")
  (vl-registry-write
        "HKEY_CURRENT_USER\\Software\\ABC\\MyDraftingSettings"
        "FilletRadius" 0
  )
  (vl-registry-write
       "HKEY_CURRENT_USER\\Software\\ABC\\MyLayerSettings"
       "NotesLayer" "Notes"
  )

  (vl-registry-descendents "HKEY_CURRENT_USER\\Software\\ABC")

(princ)
)
```



The first time the custom program `WinReg` is ran, there is no value assigned to the `Discipline` key under "`HKEY_CURRENT_USER\\Software\\ABC`" so the value of "<Missing>" is returned and displayed in the message box. Along with a value being assigned to the `Discipline` key, additional keys are also written to the Windows Registry. When the program is ran again, the value of "Civil" is returned in the message box since the `Discipline` key has an assigned value.

Xdata, Dictionaries, and Xrecords

Storing data in the Windows Registry is great but it isn't always the most practical or beneficial approach for persisting data between drawing sessions, especially if you want the data related to your custom programs to be retained with the drawing. AutoCAD allows you to:

- Store some limited information in drawing-based system variables; USERI1-5 and USERR1-5
- Create custom dictionaries and add them to the named objects dictionary of a drawing; Xrecords can then be appended to a dictionary which hold the custom information
- Attach information to individual objects with extended data or extension dictionaries

Attaching Xdata to an Individual Object

Each object in a drawing can have non-graphical data attached to it, this data is known as *extended data* or more commonly referred to as xdata. Xdata can be used to represent custom object properties or mappings to external data outside of a drawing. Before xdata is attached to an object, you must first register the name of the application to be associated with the xdata using the `regapp` function. The `regapp` function takes a single argument and that is the application name as a string; `(regapp app)`.

The following is an overview of how xdata is organized and stored in a drawing:

Drawing
- Model Space
  - Object
    - Xdata
- Symbol Tables
  - Registered Applications
    - Custom Registered Application

Here is an example of xdata attached to might look like:

```
(-3 ("ACAD" (1000 . "DSTYLE") (1002 . "{") (1070 . 347)
            (1005 . "220") (1002 . "}"))
```

Xdata must start with a DXF code of -3 followed by a list that contains the name of the associated registered application and dotted pairs that represent the data to be attached to the object. Each dotted pair must start with a DXF code of 1000 or higher that indicates the data type of the value to be stored and then the actual value to be stored. The following table lists the DXF codes and the value types that can be stored with xdata.

DXF Code	Value Type
1000-1009	String (255-character limit prior to AutoCAD 2000 and 2,049 single-byte characters with AutoCAD 2000 and later)
1010-1059	Double-precision floating-point value
1060-1070	16-bit signed integer value
1071	32-bit sign long integer value

There are several different ways of working with xdata on objects and the approach you choose will be based on whether you are more comfortable with the functions available as part of the AutoCAD COM library or legacy AutoLISP. The following table lists the functions that are available to work with xdata using the AutoCAD COM library.

Function with syntax	Description
`(vla-getxdata obj app artypes arvalues)`	Retrieves the data for an application on an object
`(vla-setxdata obj artypes arvalues)`	Sets or removes the data for an application on an object

These code examples demonstrate how to attach and remove xdata from an object using functions in the AutoCAD COM library:

```
;; Helper function for SetXdata-VL and RemoveXdata-VL
;; Usage: (xdataLISP-VL entityName "MyApp" xDataTypeCodes xDataValues)
(defun xdataLISP-VL (entityName appName xDtypeNew xDvalueNew /
                     obj dimSize dimSizeNew xDtypeOld xDvalueOld)
  (if (/= entityName nil)
    (progn
      (regapp appName)
      (setq obj (vlax-ename->vla-object entityName))
      (vla-getxdata obj appName 'xDtype 'xDvalue)
      (vla-setxdata obj xDtypeNew xDvalueNew)
    )
  )
 (princ)
)
;; Demonstrates how to attach Xdata with the xdataLISP-VL function
(defun c:SetXdata-VL ( / xDtype xDvalue)
  (setq xDtype (vlax-make-safearray vlax-vbinteger '(0 . 5)))
  (setq xDvalue (vlax-make-safearray vlax-vbvariant '(0 . 5)))
```

```
  (vlax-safearray-fill xDtype
    (list 1001 1000 1002 1070 1005 1002)
  )
  (vlax-safearray-fill xDvalue
     (list "ACAD" "DSTYLE" "{" 347
           (cdr (assoc 5 (entget (tblobjname "ltype" "center"))))
           "}")
  )

  (xdataLISP-VL (car (entsel "\nSelect object to add Xdata to: "))
                "ACAD" xDtype xDvalue
  )
)

;; Removes the attached Xdata using the xdataLISP-VL function
(defun c:RemoveXdata-VL ()
  (setq xDtype (vlax-make-safearray vlax-vbinteger '(0 . 0)))
  (setq xDvalue (vlax-make-safearray vlax-vbvariant '(0 . 0)))

  (vlax-safearray-fill xDtype (list 1001))
  (vlax-safearray-fill xDvalue (list "ACAD"))

  (xdataLISP-VL
    (car (entsel "\nSelect object to remove Xdata from: "))
    "ACAD" xDtype xDvalue
  )
)
```

Note: If you require your custom programs to be supported on both Windows and Mac OS X, you won't be able to utilize the functions available with the AutoCAD COM library.

Once xdata has been attached to an object, you can get the xdata and list the values stored using the arrays returned by `vla-getxdata` or the data returned by `entget`:

```
;; List the Xdata attached to an object
(defun c:ListXdata ()
  (assoc -3 (entget
    (car (entsel "\nSelect object to lists Xdata: ")) '("*"))
  )
)
```

These code examples demonstrate how to attach and remove xdata from an object using legacy AutoLISP functions:

```
;; Helper function for SetXdata and RemoveXdata
;; Usage: (xdataLISP entityName '("MyApp" (1004 . "Piping"))))
(defun xdataLISP (entityName lstValue / entData)
  (if (/= entityName nil)
    (progn
      (regapp "ACAD")

      (setq entData (entget entityName '("ACAD")))

      (if (/= (assoc -3 entData) nil)
        (if (= (nth 0 (cadr (assoc -3 entData))) "ACAD")
          (setq entData (subst
                          (cons -3 (list lstValue))
                          (assoc -3 entData) entData))
          (setq entData (append entData (list (list -3 lstValue))))
        )
        (setq entData (append entData (list (list -3 lstValue))))
      )

      (entmod entData)
      (entupd entityName)
    )
  )

 (princ)
)

;; Demonstrates how to attach Xdata using the xdataLISP function
(defun c:SetXdata ()
  (xdataLISP (car (entsel "\nSelect object to add Xdata to: "))
          (list "ACAD" '(1000 . "DSTYLE") '(1002 . "{") '(1070 . 347)
                (cons 1005 (cdr (assoc 5 (entget
                  (tblobjname "ltype" "center"))))) '(1002 . "}")
          )
  )
)

;; Removes the attached Xdata using the xdataLISP helper function
(defun c:RemoveXdata()
  (xdataLISP
    (car (entsel "\nSelect object to remove Xdata from: ")) '("ACAD")
  )
)
```

Storing Data in Custom Dictionaries

Like Xdata, dictionaries are used to hold non-graphical data that can be referenced by multiple different objects in a drawing. AutoCAD has been using named object dictionaries to store data related to many features; multiline styles, groups, and layer filters among others. Dictionaries can hold graphical objects and non-graphical records, commonly referred to as *xrecords*. Xrecords are similar in concept to a row in a spreadsheet or a record in a database. Along with named object dictionaries which are stored centrally in a drawing, graphical and non-graphical objects support extension dictionaries which provide a more organized way of attaching data to an object.

The following is an overview of how dictionaries and xrecords are organized and stored in a drawing:

Named Object Dictionary

- Drawing
  - Dictionaries Collection
    - Dictionary
      - Xrecord1
      - Xrecord2

Extension Dictionary

- Drawing
  - Model or Paper Space
    - Object
      - Dictionary
        - Xrecord1
        - Xrecord2

When assigning values to an xrecord, you use DXF codes just like when defining xdata but the DXF codes used fall into the normal range used for objects which is between 1 and 369. The following table lists the DXF codes used with xrecords and the value types that they can be assigned. For additional information on DXF codes look up the topic Group Codes in Numerical Order in the AutoCAD Online Help system.

DXF Code	Value Type
0-9	String (255-character limit prior to AutoCAD 2000 and 2,049 single-byte characters with AutoCAD 2000 and later)
10-39	Double precision 3D point value
40-59	Double-precision floating-point value
60-79	16-bit signed integers
90-99	32-bit sign long integers
100	String (255-character maximum string lengths)

DXF Code	Value Type
102	String (255-character maximum string lengths)
105	String representing hexadecimal value (handle)
110-119	Double precision floating-point value
120-129	Double precision floating-point value
130-139	Double precision floating-point value
140-149	Double precision scalar floating-point value
170-179	16-bit integer value
210-239	Double-precision floating-point value
270-279	16-bit integer value
280-289	16-bit integer value
290-299	Boolean flag value
300-309	Arbitrary text string
310-319	String representing hex value of binary chunk
320-329	String representing hex handle value
330-369	String representing hex object IDs

Since dictionaries are stored in the drawing differently than xdata and contain xrecords, they require you to utilize a different set of functions from those that you have already seen with xdata. The following table lists the functions that are available to work with xrecords and dictionaries using the AutoCAD COM library.

Function with syntax	Description
`(vla-get-dictionaries document)`	Retrieves a collection of all named dictionaries stored in a drawing
`(vla-addxrecord dict-obj name)`	Adds an xrecord to a dictionary object
`(vla-setxrecorddata xrec-obj types values)`	Sets the data for an xrecord
`(vla-getxrecorddata xrec-obj types values)`	Gets the data for an xrecord

```
;; Creates a dictionary using Visual LISP
(defun c:CreateDictionary-VL ( / ); acadObj docObj dictsColl)
  (setq acadObj (vlax-get-acad-object))
  (setq docObj (vla-get-activedocument acadObj))
  (setq dictsColl (vla-get-dictionaries docObj))
```

```
  (setq dictObj (vla-add dictsColl "MY_CUSTOM_DICT2"))
  (setq xrecObj (vla-addxrecord dictObj "XREC_1"))

  (setq xDtype (vlax-make-safearray vlax-vbinteger '(0 . 2)))
  (setq xDvalue (vlax-make-safearray vlax-vbvariant '(0 . 2)))
  (setq xPoint (vlax-make-safearray vlax-vbdouble '(0 . 2)))

  (vlax-safearray-fill xPoint '(5.0 5.0 0.0))
  (vlax-safearray-fill xDtype '(1 10 71))
  (vlax-safearray-fill xDvalue
    (list "Visual LISP Dictionary" xPoint 11)
  )

  (vla-setxrecorddata xrecObj xDtype xDvalue)

 (princ)
)

;; Deletes a dictionary using Visual LISP
(defun c:DeleteDictionary-VL ( / acadObj docObj dictsColl dictObj)
  (setq acadObj (vlax-get-acad-object))
  (setq docObj (vla-get-activedocument acadObj))

  (setq dictsColl (vla-get-dictionaries docObj))
  (vla-delete (vla-item dictsColl "MY_CUSTOM_DICT2"))

 (princ)
)

;; Prints the data of the Xrecord using Visual LISP
(defun c:PrintXrec-VL ( / acadObj docObj dictsColl
                          dictObj xrecObj xDtype xDvalue)
  (setq acadObj (vlax-get-acad-object))
  (setq docObj (vla-get-activedocument acadObj))

  (setq dictsColl (vla-get-dictionaries docObj))
  (setq dictObj (vla-item dictsColl "MY_CUSTOM_DICT2"))
  (setq xrecObj (vla-item dictObj "XREC_1"))

  (vla-getxrecorddata xrecObj 'xDtype 'xDvalue)

  (if (/= xDtype nil)
    (progn
      (princ (vlax-safearray->list xDtype))
      (terpri)
      (princ (vlax-safearray->list xDvalue))
    )
  )

 (princ)
)
```

The following table lists the legacy AutoLISP functions that can be used to work with dictionaries.

Function with syntax	Description
`(dictadd dict-entity rec-name entity)`	Adds a custom dictionary to the specified entity
`(dictremove entity name)`	Removes a custom dictionary
`(dictnext entity [flag])`	Returns the next entry in a dictionary
`(dictrename entity old-name new-name)`	Renames a dictionary from the old name to a new name
`(dictsearch entity name [flag])`	Searches a dictionary for an entry

```
;; Demonstrates how to create a dictionary
(defun c:CreateDictionary ( / dict xname newdict datalist)
  (setq dict (list '(0 . "DICTIONARY") '(100 . "AcDbDictionary")))
  (setq xname (entmakex dict))

  (setq newdict (dictadd (namedobjdict) "MY_CUSTOM_DICT" xname))

  (setq datalist
       (append (list '(0 . "XRECORD") '(100 . "AcDbXrecord"))
                     '((1 . "Custom string") (10 5.0 5.0 0.0)
                         (71 . 11))))

  (setq xname (entmakex datalist))
  (dictadd newdict "XREC_1" xname)

 (princ)
)

;; Deletes a custom dictionary
(defun c:DeleteDictionary ()
  (dictremove (namedobjdict) "MY_CUSTOM_DICT")
)

;; Prints the data of the Xrecord
(defun c:printXrec ( / dictName dictEntry newdictlist xrec)
  (setq dictName "MY_CUSTOM_DICT"
        dictEntry "XREC_1"
  )

  (setq newdictlist (dictsearch (namedobjdict) dictName))
  (setq xrec (cadr (member (cons 3 dictEntry) newdictlist)))

  (princ (entget (cdr xrec)))

 (princ)
)
```

3 Monitoring Activity in AutoCAD with Reactors

AutoCAD monitors a lot of activities that take place in the application, while a drawing is open, and when objects are added to or modified in a drawing. The Visual LISP extension allows you to step in and catch many of these activities through what are known as *reactors*. Reactors are also known as *events* or *event handlers* in other programming languages.

Reactors come in a variety of types, some of the more common reactor types are listed in the following table.

Reactor	Description
`:VLR-AcDb-Reactor`	Database reactor
`:VLR-Command-Reactor`	Command reactor
`:VLR-DeepClone-Reactor`	Deep clone reactor
`:VLR-DocManager-Reactor`	Document management reactor
`:VLR-DWG-Reactor`	Drawing reactor (opening or closing a drawing file)
`:VLR-DXF-Reactor`	DXF file handling reactor (reading and writing)
`:VLR-Editor-Reactor`	General editor reactor (available for backwards compatibility)
`:VLR-Insert-Reactor`	Block insertion reactor
`:VLR-Linker-Reactor`	Linker reactor
`:VLR-Lisp-Reactor`	LISP reactor
`:VLR-Miscellaneous-Reactor`	Reactors that are not part of other reactor types (pick first changes and layout switching)
`:VLR-Mouse-Reactor`	Mouse reactor (right-click and double-click)
`:VLR-Object-Reactor`	Object reactor (appending and modifying objects)
`:VLR-SysVar-Reactor`	System variable reactor
`:VLR-Toolbar-Reactor`	Toolbar reactor (toolbar images change size)
`:VLR-Undo-Reactor`	Undo reactor
`:VLR-Wblock-Reactor`	Writing a block reactor
`:VLR-Window-Reactor`	AutoCAD or drawing window reactor (window is moved or resized)
`:VLR-XREF-Reactor`	XREF reactor (attaching, reloading, or detaching)

For more information about reactors, look at the [About Reactor Types and Events (AutoLISP/ActiveX)] in the AutoCAD Online Help system.

Important: When using reactors, you can't call a command using the `Command` function; this is due to the way reactors are designed. You must use the functions of the Visual LISP extension and the AutoCAD COM library to mimic the behavior of the commands you might want to use.

Command Reactor

Command reactors are one of the most commonly used reactors and they allow you to be notified of when a command starts, ends, gets cancelled, or fails.

Note: While custom AutoLISP function names prefixed with `c:` can be entered at the Command prompt, AutoCAD doesn't recognize them as commands, so they can't be monitored using command related reactors. However, the `vlax-add-cmd` function can be used to register a custom AutoLISP function as a command so it can be monitored with command related reactors.

The following sample code demonstrates how to watch for the use of the Qdim, Hatch, Batch, and Gradient commands and perform a task before the command starts and do something after the command ends.

```
;; Check to see if our custom command reactors
;; have been loaded into the current drawing
(if (= hyp-rctCmds nil)
  (setq hyp-rctCmds (vlr-command-reactor nil
                          '((:vlr-commandCancelled . hyp-cmdAbort)
                            (:vlr-commandEnded . hyp-cmdAbort)
                            (:vlr-commandFailed . hyp-cmdAbort)
                            (:vlr-commandWillStart . hyp-cmdStart)
                           )
                    )
  )
)

;; Callback used when the user presses ESCape
;; or when the command ends by itself or due to a problem
(defun hyp-cmdAbort (param1 param2)
  (if (/= hyp-gClayer nil)
    (setvar "clayer" hyp-gClayer)
  )

  (setq hyp-gClayer nil)
)

;; Callback used when a command is started
(defun hyp-cmdStart (param1 param2 / acadObj docObj layerObj)

  (setq hyp-gClayer (getvar "clayer"))

  (cond
    ((= (car param2) "QDIM")(prompt "\nQDIM started"))
    ((= (car param2) "NUMMODAL")(prompt "\nCustom Command started"))
    ((or (= (car param2) "HATCH")
       (= (car param2) "BHATCH")
       (= (car param2) "GRADIENT")
     )
      (progn
       (if (= (tblsearch "layer" "Hatch") nil)
         (progn
```

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

            (setq docObj (vla-get-activedocument acadObj))

            (setq layerObj (vla-add (vla-get-layers docObj) "Hatch"))

           (vla-put-color layerObj acRed)
         )
       )
       (setvar "clayer" "Hatch")
     )
   )
 )
)

; Defines a function that displays the number of objects selected
(defun NumberOfObjects ( / ss)
  (setq ss (ssget))
  (alert (strcat "Number of objects selected: " (itoa (sslength ss))))
)

; Adds a modal command based on the custom function NumberOfObjects
(vlax-add-cmd "NumModal" 'NumberOfObjects
              "NumModal" ACRX_CMD_MODAL)

; Removes the command
(vlax-remove-cmd "NumModal")
```

4 Open and Close External Drawings

AutoLISP is commonly limited to working with objects in the current drawing, but the AutoCAD COM library allows you to access other open drawings in the AutoCAD application. Along with the AutoCAD COM library, AutoCAD also ships with a library named AutoCAD/ObjectDBX Common.

The AutoCAD/ObjectDBX Common library allows you to work with a drawing without opening the file in the application, this can make it more efficient to manipulate and access the contents of a drawing. When a drawing is opened using ObjectDBX, your custom program can't prompt the user to select objects or specify points using the mouse in the drawing.

Before you can utilize ObjectDBX, you must first import and expose the functions of the ObjectDBX library to the AutoCAD program. The `vlax-import-type-library` function is used to import a COM library into AutoCAD, and the `vlax-create-object` function is used to create an instance of an object available as part an imported library.

Note: If an instance of an object is created using `vlax-create-object`, it is good practice to remove the object from memory using the `vlax-release-object` function.

The following sample code demonstrates how to import the ObjectDBX library, and then proceed to open a drawing in-memory and from there copy all the dimension styles from the in-memory drawing into the current drawing.

```
;; Open drawing in the background and import all dimension styles
(defun c:AccessExternalDrawing ( / acdbObj cntDimstyles acadObject
                                   arTemp arDimStyles cntStep)

  (if (= acLibImport nil)
    (progn
      (vlax-import-type-library :tlb-filename
          "C:\\Program Files\\Common Files\\
               Autodesk Shared\\axdb23enu.tlb"
                                :methods-prefix "acdbm-"
                                :properties-prefix "acdbp-"
                                :constants-prefix "acdbc-"
      )
      (setq acLibImport T)
    )
  )

  (setq acdbObj (vlax-create-object "ObjectDBX.AxDbDocument.23"))

  (acdbm-open acdbObj
       "C:\\Datasets\\Lee Ambrosius\\SD196926\\External Drawing.dwg")

  (setq cntDimstyles 0)

  (vlax-for acadObject (vla-get-Dimstyles acdbObj)

    (setq arTemp arDimStyles)
    (setq arDimStyles (vlax-make-safearray vlax-vbObject
                                           (cons 0 cntDimstyles)))

    (setq cntStep 0)
    (repeat cntDimstyles
      (vlax-safearray-put-element
          arDimStyles cntStep
          (vlax-safearray-get-element arTemp cntStep)
      )
      (setq cntStep (1+ cntStep))
    )

    (vlax-safearray-put-element arDimStyles cntDimstyles acadObject)
    (setq cntDimstyles (1+ cntDimstyles))
  )

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

  (setq docObj (vla-get-activedocument acadObj))

  (setq dbObj (vla-get-database docObj))

  (setq dimstylesColl (vla-get-dimstyles dbObj))

  (vla-copyobjects acdbObj arDimStyles dimstylesColl)
```

```
  (vlax-release-object acdbObj)
 (princ)
)
```

5 Working with Windows

Microsoft creates many different programming libraries related to Windows and other applications they develop not only for themselves, but also for other developers to take advantage. Since the Visual LISP extension allows you to tap into COM libraries outside of AutoCAD, you can access some of the Windows related libraries to manipulate the file system, collect information about the current user and workstation among many other things.

Two of the libraries that Microsoft provides for working with Windows are known as the Windows Host Scripting Object and Microsoft Scripting Runtime libraries. These libraries allow you to:

- Create file shortcuts
- Access special named folders; Documents, Pictures, …
- Query and set environment variables
- Access the drive, folder, and file structures of Windows

Create Desktop Shortcut for AutoCAD

Desktop shortcuts can be created with the `WScript.Shell` object in the Windows Host Scripting Object library. A desktop shortcut can be used to customize and standardize the way AutoCAD starts up. For example, you can use the `/w` command line switch to set a workspace current, and other command line switches such as `/t` to specify which template file should be used for the default drawing and `/p` to set a user profile current.

The following code example demonstrates how to create a desktop shortcut that sets the 3D Modeling workspace current during start up.

```
;; Create shortcut on desktop
(defun c:CreateDesktopShortcut ( / wshShell desktopFldr
                                   myDocsFldr shrtObj)

  (if (= wshLibImport nil)
    (progn
      (vlax-import-type-library
         :tlb-filename "c:\\windows\\system32\\wshom.ocx"
         :methods-prefix "wshm-"
         :properties-prefix "wshp-"
         :constants-prefix "wshk-"
      )
      (setq wshLibImport T)
    )
  )

  (setq wshShell (vlax-create-object "WScript.Shell"))
```

```
  (setq desktopFldr
    (wshm-Item
       (wshp-get-SpecialFolders wshShell) "Desktop"))

  (setq myDocsFldr
    (wshm-Item
       (wshp-get-SpecialFolders wshShell) "MyDocuments"))

  (setq shrtObj
    (wshm-CreateShortcut
       wshShell (strcat desktopFldr "\\My AutoCAD.lnk")))

  (wshp-put-TargetPath shrtObj
       (strcat "\"" (vla-get-FullName (vlax-get-acad-object)) "\""))

  (wshp-put-Arguments shrtObj "/w \"3D Modeling\"")

  (wshp-put-Description shrtObj "Custom AutoCAD Desktop Shortcut")

  (wshp-put-WindowStyle shrtObj wshk-WshNormalFocus)

  (wshp-put-HotKey shrtObj "Ctrl+Alt+A")

  (wshp-put-WorkingDirectory shrtObj myDocsFldr)

  (wshp-put-IconLocation shrtObj
       (strcat (vla-get-FullName (vlax-get-acad-object)) ", 0"))

  (wshm-save shrtObj)

  (vlax-release-object wshShell)
 (princ)
)
```

You can learn more about the command line switches available by reading the Command Line Switch Reference topic in the AutoCAD Online Help system. For additional information on the Windows Host Script Object, visit Microsoft's website and read the Windows Script Documentation.

Working with Environment Variables

Environment variables are often used to store information about the current user logged into the workstation, and on a limited basis hardware and software settings. Most modern applications utilize the Windows registry for storing values, but environment variables are an efficient way to share values across applications. Some environment variables can be retrieved or set using AutoLISP the `setenv` and `getenv` functions, but these functions are mainly limited to accessing a subset of AutoCAD settings stored in the Windows Registry and common Windows environment variables.

Note: When using the `setenv` and `getenv` functions, environment variable names are case sensitive.

The following are examples of accessing the `MaxHatch` environment variables which controls the maximum number of lines that can be displayed in a hatch pattern.

```
(getenv "MaxHatch")
"100000"

(setenv "MaxHatch" "500000")
"500000"
```

By using the Windows Host Scripting Object, you can access all defined Windows environment variables rather than just a limited set and resolve strings with expandable variables. By having access to all Windows environment variables, you can work with and store custom settings outside of the Windows Registry that a variety of applications can access.

Note: Environment variables can be defined as user or machine specific. You use the keyword "USER" when you want to work with variables that are specific to the user that is logged in or "SYSTEM" to work with variables that are machine specific and not user specific.

To see which variables are specific to you as a user or the system, display the Windows Control Panel. In the Control Panel window, if View By is set to Category, click System and Security. Click System and then click Advanced System Settings on the left. In the System Properties dialog box, Advanced tab, click Environment Variables. In the Environment Variables dialog box, add, edit or remove the variables as desired.

Tip: You can also see which variables are available to you by launching the Windows Command Prompt window and enter the command `SET`.

The following code examples demonstrate how to expand a text string that contains a Windows environment variable and manipulate the value of a Windows environment variable.

```
;; Shows how to use expanding environment strings
;; Usage: (ExpEnvStr "%TEMP%\\MYDATA")
;; Results of sample: "C:\\DOCUME~1\\Lee\\LOCALS~1\\Temp\\MYDATA"
(defun ExpEnvStr (strVal / wshShell strValRet)
  (setq wshShell (vlax-create-object "WScript.Shell"))

  (setq strValRet (vlax-invoke-method
                      wshShell
                      'ExpandEnvironmentStrings
                      strVal
                  ))

  (vlax-release-object wshShell)

  strValRet
)

;; Retrieve the value of the environment variable
;; Usage (1): (GetEnvStr "SYSTEM" "USERID")
;; Usage (2): (GetEnvStr "SYSTEM" "PROCESSOR_ARCHITECTURE")
(defun GetEnvStr (strVarType strVarName / wshShell envVars strValRet)
  (setq wshShell (vlax-create-object "WScript.Shell"))

  (setq envVars (vlax-get-property wshShell 'Environment strVarType))

  (setq strValRet (vlax-get-property envVars 'Item strVarName))
```

```
  (vlax-release-object wshShell)

  strValRet
)

;; Set the value to an environment variable
;; Usage: (SetEnvStr "SYSTEM" "USERID" "L123")
(defun SetEnvStr (strVarType strVarName strVarVal / wshShell envVars)
  (setq wshShell (vlax-create-object "WScript.Shell"))

  (setq envVars (vlax-get-property wshShell 'Environment strVarType))

  (vlax-put-property envVars 'Item strVarName strVarVal)

  (vlax-release-object wshShell)
 (princ)
)
```

Listing Available Drives

Access to the Windows file system can be gained using the `Scripting.FileSystemObject` object in the Windows Scripting Run-time library. The file system can be used to help locate client files and learn more about how a workstation is configured or the programs that are installed.

The following code example demonstrates how to list all available drives and the files/folders of the root folder on each drive.

```
;; Load the Windows Scripting Run-time Library
(if (= wsrLibImport nil)
  (progn
    (vlax-import-type-library
         :tlb-filename "c:\\windows\\system32\\scrrun.dll"
         :methods-prefix "wsrm-"
         :properties-prefix "wsrp-"
         :constants-prefix "wsrk-"
    )
    (setq wsrLibImport T)
  )
)

(defun c:ListDrives ( / wsrFSO wsrDrives wsrDrive)
  ;; Create reference to File System Object
  (setq wsrFSO (vlax-create-object "Scripting.FileSystemObject"))

  ;; Get the Drives collection
  (setq wsrDrives (wsrp-get-drives wsrFSO))

  ;; Step through each of the available drives
  (vlax-for wsrDrive wsrDrives
    (prompt (strcat "\n>" (wsrp-get-DriveLetter wsrDrive)))
    (prompt (strcat "\n  Path: " (wsrp-get-Path wsrDrive)))
    (prompt (strcat "\n  Free Space: "
```

```
               (rtos (vlax-variant-value
                       (wsrp-get-FreeSpace wsrDrive)) 2 0) " bytes"))

    (prompt "\n  Files:")
    (vlax-for wsrFile (wsrp-get-Files (wsrp-get-RootFolder wsrDrive))
      (prompt (strcat "\n    " (wsrp-get-name wsrFile)))
    )

    (prompt "\n  Folders:")
    (vlax-for wsrFolder (wsrp-get-SubFolders
                          (wsrp-get-RootFolder wsrDrive))
      (prompt (strcat "\n    " (wsrp-get-name wsrFolder)))
    )
  )

  ;; Release the Run-time Scripting Object
  (vlax-release-object wsrFSO)
 (princ)
)
```

For additional information on the Microsoft Scripting Run-time, visit Microsoft's website and read the Scripting Run-Time Reference.

6 Working with Microsoft Office

Being able to manipulate objects in AutoCAD and Windows can go a long way to improving workflows and help to decrease the number of redundant tasks that you might normally perform every day. Being able to efficiently use design information downstream can help a project reach completion faster and with greater accuracy. As previously mentioned, the Visual LISP extension provides the capabilities to work with other COM libraries and Microsoft offers many different COM libraries that allow you to extend the functionality of its Microsoft Office applications, such as Word and Excel.

The following code examples demonstrate

- Printing a Microsoft Word document to the default printer
- Showing how to take information from AutoCAD to create a Microsoft Word document
- Pushing and pulling information from an Microsoft Excel spreadsheet to modify content in a drawing
- Accessing information from a Microsoft Access (MDB) database

For information on the COM libraries available for Microsoft Office applications, see the Office VBA Reference and ADO API Reference on Microsoft's website.

Print Documents through Microsoft Word

If you create documents with Microsoft Word as part of a bid package, you will most likely want to ensure they are printed and ready for delivery. Whatever your reason might be, you can open and print a Word document off to streamline your processes.

The following code example shows how to create and print a Word (DOC/DOCx) file.

```
;; Usage: (PrintMSWordDoc
;;             "c:\\datasets\\lee ambrosius\\sd196926\\spec1.doc")
```

```
;; Open a document in MS Word and print it using the default printer
(defun PrintMSWordDoc (strWordDoc / wordObj wordDocsObj wordDocObj)
  ;; Load the Word Library
  (if (= wordLibImport nil)
    (progn
      (vlax-import-type-library
           :tlb-filename (strcat strMSOfficePath "MSWORD.OLB")
           :methods-prefix "wordm-"
           :properties-prefix "wordp-"
           :constants-prefix "wordc-"
       )
      (setq wordLibImport T)
    )
  )

  ;; Create a reference to the Database engine
  (setq wordObj (vlax-create-object
                    (strcat "Word.Application." strOfficeVer)))

  ;; Get a reference to the Docuemnts collection in Word
  (setq wordDocsObj (vlax-get-property wordObj 'Documents))

  ;; Open the drawing
  (setq wordDocObj (wordm-open wordDocsObj strWordDoc))

  ;; Print it out
  (if (/= (vlax-get-property wordObj 'ActivePrinter)
          "Microsoft Print to PDF")
    (wordm-printout wordDocObj)
    (wordm-printout wordDocObj
          :vlax-true
          :vlax-false
          wordc-wdPrintAllDocument
          (strcat (substr
                      strWordDoc
                      1
                      (- (strlen strWordDoc)
                         (strlen (vl-filename-extension strWordDoc)))
                  )
                  ".pdf"
          )
    )
  )

  ;; Close the document and don't save any changes
  (wordm-close wordDocObj wordc-wdDoNotSaveChanges)
```

```
  ;; Close the instance of the MS Word application
  (wordm-quit wordObj wdDoNotSaveChanges)

  ;; Release the Word App object
  (vlax-release-object wordObj)
 (princ)
)
```

Create new Word Document Based on Information in Drawing

Often when you create a drawing, you might also need to generate a quote based on the information contained in the drawing. Using the Microsoft Word COM library you can create a new document and populate the document with content based on information in a drawing.

The following code example demonstrates how to create a new document from scratch and create a series of paragraphs that contain the names of the layers, dimension styles, text styles and blocks contained in the drawing.

```
;; Creates a new Word document and extracts symbol table names
;; from the current drawing and lists them in the new Word document.
(defun c:CreateMSWordDoc ( / acadObj docObj wordObj wordDocsObj
                             wordDocObj wordParasObj
                             wordParaObj wordParaRangeObj
                             wordSentencesObj wordSentenceObj
                             wordSentenceFontObj)

  ;; Load the Word Library
  (if (= wordLibImport nil)
    (progn
      (vlax-import-type-library
           :tlb-filename (strcat strMSOfficePath "MSWORD.OLB")
           :methods-prefix "wordm-"
           :properties-prefix "wordp-"
           :constants-prefix "wordc-"
      )
      (setq wordLibImport T)
    )
  )

  ;; Create a reference to AutoCAD
  (setq acadObj (vlax-get-acad-object))

  ;; Create a reference to the current drawing
  (setq docObj (vla-get-activedocument acadObj))

  ;; Create a reference to the Word Application
  (setq wordObj (vlax-create-object
                    (strcat "Word.Application." strOfficeVer)))

  ;; Get the Documents collection and add a new document
  (setq wordDocsObj (vlax-get-property wordObj 'Documents))
  (setq wordDocObj (wordm-Add wordDocsObj))
```

```
  ;; Get the Paragraphs collection
  (setq wordParasObj (wordp-get-paragraphs wordDocObj))

  ;; Add a new paragraph and range
  (setq wordParaObj (wordm-add wordParasObj))
  (setq wordParaRangeObj (wordp-get-range wordParaObj))

  ;; Add text to the paragraph/range
  (wordp-put-text wordParaRangeObj
    (strcat "Welcome to Autodesk University 2018"
            "\nWhat you are seeing is a very basic "
            "example of creating a Word document using "
            "the MS Word object. The example code creates "
            "a listing of layers, dimension styles, text "
            "styles and block names from the current drawing.\n\n"))

  ;; Get the first sentence
  (setq wordSentencesObj (wordp-get-sentences wordParaRangeObj))
  (setq wordSentenceObj (wordm-item wordSentencesObj 1))

  ;; Format the sentence: Bold, Underline, and 16 points
  (setq wordSentenceFontObj (wordp-get-font wordSentenceObj))
  (wordp-put-bold wordSentenceFontObj :vlax-true)
  (wordp-put-underline wordSentenceFontObj :vlax-true)
  (wordp-put-size wordSentenceFontObj 16)

  ;; Add a new paragraph and range
  (setq wordParaObj (wordm-add wordParasObj))
  (setq wordParaRangeObj (wordp-get-range wordParaObj))

  ;; Add text to the paragraph/range
  (wordp-put-text wordParaRangeObj
    (strcat "Report from drawing: "
         "\n\t" (vla-get-Path docObj)
         "\\" (vla-get-Name docObj) "\n\n"))

  ;; Step through and report the layers, dim styles, text styles,
  ;; and blocks in the current drawing
  (listItems (vla-get-layers docObj) "Layer List")
  (listItems (vla-get-dimstyles docObj) "Dimension Style List")
  (listItems (vla-get-textstyles docObj) "Text Style List")
  (listItems (vla-get-blocks docObj) "Block List")

  ;; Make the instance of the application visible
  (wordp-put-visible wordObj :vlax-true)

  (vlax-release-object wordObj)
 (princ)
)
```

```
;; Used to create a new paragraph based on the items in the collection
;; The title line is formatted as bold, underlined and a size of 14.
(defun listItems (coll title / itemList wordParaObj wordParaRangeObj
                               wordSentencesObj wordSentenceFontObj)
  (setq itemList "")
  ;; Step through the symbol table collection in the drawing
  (vlax-for for-item coll
    (setq itemList (strcat itemList "\n" (vla-get-name for-item)))
  )

  ;; Add a new paragraph and range
  (setq wordParaObj (wordm-add wordParasObj))
  (setq wordParaRangeObj (wordp-get-range wordParaObj))

  ;; Add text to the paragraph/range
  (wordp-put-text wordParaRangeObj (strcat title itemList "\n\n"))

  ;; Get the first sentence
  (setq wordSentencesObj (wordp-get-sentences wordParaRangeObj))
  (setq wordSentenceObj (wordm-item wordSentencesObj 1))

  ;; Format the sentence: Bold, Underline, and 14 points
  (setq wordSentenceFontObj (wordp-get-font wordSentenceObj))
  (wordp-put-bold wordSentenceFontObj :vlax-true)
  (wordp-put-underline wordSentenceFontObj :vlax-true)
  (wordp-put-size wordSentenceFontObj 14)
)
```

Extract Information to Microsoft Excel Workbook

Microsoft Excel is a very powerful application that allows you to perform complex calculations or even create tables of information such as door and window schedules. The following code sample demonstrates how you might use an Excel workbook to store extracted information and then later update the original drawing based on the values in the workbook. In this sample, the diameter values of all circles are extracted along with their handles. The object handle is used as a unique key in the workbook and a way to map the values back into the drawing.

```
;; Creates a new Excel spreadsheet and extracts the circles
;; from the current drawing. The extracted information is added
;; to the new Excel spreadsheet.
(defun c:ExtractDrawingToExcel ( / acadObj docObj excelObj
                                   excelWorkbooksObj excelWorkbookObj
                                   excelWorksheetsObj
                                   excelWorksheetObj
                                   excelRangeObj spaceObj
                                   cnt for-item)
  ;; Import the Excel library
  (if (= excelLibImport nil)
    (progn
      (vlax-import-type-library
           :tlb-filename (strcat strMSOfficePath "Excel.exe")
           :methods-prefix "exlm-"
           :properties-prefix "exlp-"
```

```
          :constants-prefix "exlc-"
     )
     (setq excelLibImport T)
   )
)

;; Create a reference to AutoCAD
(setq acadObj (vlax-get-acad-object))

;; Create a reference to the current drawing
(setq docObj (vla-get-activedocument acadObj))

;; Create a reference to the Excel Application
(setq excelObj (vlax-create-object
                (strcat "Excel.Application." strOfficeVer)))

;; Make the Excel application visible
(exlp-put-visible excelObj :vlax-true)

;; Get the Wrokbooks collection and create a new workbook
(setq excelWorkbooksObj (vlax-get-property excelObj 'Workbooks))
(setq excelWorkbookObj (exlm-add excelWorkbooksObj))

;; Get the Worksheets collection and get the first worksheet
(setq excelWorksheetsObj
          (vlax-get-property excelWorkbookObj 'Worksheets))
(setq excelWorksheetObj (exlp-get-item excelWorksheetsObj 1))

; Get the first cell and add text to it
(setq excelRangeObj (exlp-get-range excelWorksheetObj "A1"))
(exlp-put-value2 excelRangeObj
          (vlax-make-variant "Welcome to AU 2018" vlax-vbstring))

;; Get a reference to the current space in the drawing
(if (= (vla-get-activespace docObj) acModelSpace)
  (setq spaceObj (vla-get-modelspace docObj))
  (setq spaceObj (vla-get-paperspace docObj))
)

;; Add column headers
(setq excelRangeObj (exlp-get-range excelWorksheetObj "A2"))
(exlp-put-value2 excelRangeObj
                 (vlax-make-variant "Handle" vlax-vbstring))

(setq excelRangeObj (exlp-get-range excelWorksheetObj "B2"))
(exlp-put-value2 excelRangeObj
                  (vlax-make-variant "Radius" vlax-vbstring))

;; Step through each of the objects in the current space
;; and extract only the circles
(setq cnt 3)
```

```
  ;; Step through all objects in the current space
  (vlax-for for-item spaceObj
    ;; Check to see if the object is a circle
    (if (= (vla-get-objectname for-item) "AcDbCircle")
      (progn
        ;; If the object is a circle, get its handle and radius,
        ;; and then add the values to columns A and B
        (setq excelRangeObj (exlp-get-range excelWorksheetObj
                                    (strcat "A" (itoa cnt))))
        (exlp-put-value2 excelRangeObj
                    (vlax-make-variant
                  (vla-get-handle for-item) vlax-vbstring))

        (setq excelRangeObj (exlp-get-range excelWorksheetObj
                                    (strcat "B" (itoa cnt))))
        (exlp-put-value2 excelRangeObj
                    (vlax-make-variant
                  (vla-get-radius for-item) vlax-vbdouble))
        (setq cnt (+ cnt 1))
      )
    )
  )

  ;; Save the workbook
  (exlm-saveas excelWorkbookObj
       "c:\\datasets\\lee ambrosius\\sd196926\\circle-radius"
       exlc-xlNormal "" "" :vlax-false
       :vlax-true exlc-xlNoChange exlc-xlLocalSessionChanges
       :vlax-false "" "" :vlax-false)

  (vlax-release-object excelObj)
 (princ)
)

;; Opens an Excel spreadsheet and updates the entities in the
;; drawing based on the values in the spreadsheet.
(defun c:UpdateDrawingFromExcel ( / acadObj docObj excelObj
                                    excelWorkbooksObj excelWorkbookObj
                                    excelWorksheetsObj
                                    excelWorksheetObj
                                    excelRangeObj strHandle
                                    ss cnt)
  (if (= excelLibImport nil)
    (progn
      (vlax-import-type-library
           :tlb-filename (strcat strMSOfficePath "Excel.exe")
           :methods-prefix "exlm-"
           :properties-prefix "exlp-"
           :constants-prefix "exlc-"
      )
      (setq excelLibImport T)
    )
```

```
  )

  ;; Create a reference to AutoCAD
  (setq acadObj (vlax-get-acad-object))

  ;; Create a reference to the current drawing
  (setq docObj (vla-get-activedocument acadObj))

  ;; Create a reference to the Excel Application
  (setq excelObj (vlax-create-object
                  (strcat "Excel.Application." strOfficeVer)))

  ;; Make the Excel application visible
  (exlp-put-visible excelObj :vlax-true)

  ;; Get the Workbooks collection and open the Circle-Radius file
  (setq excelWorkbooksObj (vlax-get-property excelObj 'Workbooks))
  (setq excelWorkbookObj
         (exlm-open excelWorkbooksObj
              "c:\\datasets\\lee ambrosius\\sd196926\\circle-radius"))

  ;; Get the Worksheets collection and get the first worksheet
  (setq excelWorksheetsObj
            (vlax-get-property excelWorkbookObj 'Worksheets))
  (setq excelWorksheetObj (exlp-get-item excelWorksheetsObj 1))

  ;; Get the first cell with data
  (setq excelRangeObj (exlp-get-range excelWorksheetObj "A3"))
  (setq cnt 3)

  ;; Step through the cells in the first column
  (while (setq strHandle
                (vlax-variant-value (exlp-get-value2 excelRangeObj)))
    ;; Convert the handle to an object which is a circle
    (setq ss (handent strHandle))
    (setq obj (vlax-ename->vla-object ss))

    ;; Update the circle's radius
    (setq excelRangeObj
           (exlp-get-range excelWorksheetObj (strcat "B" (itoa cnt))))
    (vla-put-radius obj
           (vlax-variant-value (exlp-get-value2 excelRangeObj)))

    ;; Step to the next row
    (setq cnt (+ cnt 1))
    (setq excelRangeObj
           (exlp-get-range excelWorksheetObj (strcat "A" (itoa cnt))))
  )

  (vlax-release-object excelObj)
 (princ)
)
```

Work with Access Database using ADO

The Microsoft ActiveX Data Objects (ADO) library allows you to access data in a Microsoft Access database. A database can be a great way to:

- Manage project and client information which can then be used to populate a title block
- Generating project cost estimates
- Manage building and IT assets

Note: If you are using Microsoft Access 2007 or earlier, you must work with the Microsoft DAO (Data Access Object) library to access data in a Microsoft Access (MDB) database. A code sample demonstrating how to access an MDB file using the DAO library can be found in the dataset for this session.

The following code sample demonstrates how to utilize an Microsoft Access (MDB) database that contains a single table and to step through the records contained in a table named `tblEmployees` as well as update a field in one of the records.

```
;; Opens an Access Database file and reports on the number of records
;; in a table and the records contained in the table.
 (defun c:AccessDatabaseADO ( / daoObj dbObj rstObj fieldsObj)

  ;; Load the ADO library
  (if (= adoLibImport nil)
    (progn
      (vlax-import-type-library
        :tlb-filename (strcat (ExpEnvStr "%PROGRAMFILES%")
                      "\\Common Files\\System\\ado\\msado15.dll")
          :methods-prefix "adom-"
          :properties-prefix "adop-"
          :constants-prefix "adoc-"
      )
      (setq adoLibImport T)
    )
  )

  ;; Create a recordset and connection object
  (setq adoRecordset (vlax-create-object "ADODB.Recordset"))
  (setq adoConnection (vlax-create-object "ADODB.Connection"))

  (setq strConnection
        (strcat "Provider=Microsoft.ACE.OLEDB.12.0;"
                "Data Source=c:\\datasets\\lee
                             ambrosius\\sd196926\\AU2018.mdb;"
                "Jet OLEDB:Database Password=;"))

  ;; Open the database
  (adom-open adoConnection strConnection "" nil nil)

  ;; Open the table
  (adom-open adoRecordset "tblEmployees"
                          adoConnection
                          adoc-adOpenKeyset
                          adoc-adLockOptimistic
```

```
                          adoc-adCmdTable)

;; Step to the first and then last record to get the
;; number of records in the table
(adom-MoveFirst adoRecordset)
(adom-MoveLast adoRecordset)
(prompt (strcat "\n"
           (itoa (adop-get-RecordCount adoRecordset))
           " records in table."))

;; Step back to the first record
(adom-MoveFirst adoRecordset)
(prompt "\n\nEmployee names")

;; Report each of the records that are in the table
(while (= (adop-get-eof adoRecordset) :vlax-false)
  (setq fieldsObj (adop-get-fields adoRecordset))

  (prompt (strcat "\nFirst name: "
             (vlax-variant-value
               (adop-get-value
              (adop-get-item fieldsObj "FirstName")))))
  (prompt (strcat "\nLast name: "
             (vlax-variant-value
               (adop-get-value
              (adop-get-item fieldsObj "LastName")))))
  (terpri)

  ;; Move to the next record
  (adom-movenext adoRecordset)
)

;; Do a search for the employee with the first name Bob
(adom-movefirst adoRecordset)
(adom-Find adoRecordset "FirstName = 'Bob'"
                        0 adoc-adSearchForward 1)

;; If the record is found them return some information
;; about the record
(if (= (adop-get-eof adoRecordset) :vlax-true)
  (prompt "\nNo matching record")
  (progn
    (setq fieldsObj (adop-get-fields adoRecordset))
    (prompt (strcat "\nBob was in room "
               (vlax-variant-value
              (adop-get-value
                (adop-get-item fieldsObj "RoomNumber")))))
    (terpri)

    ;; Change the value of the RoomNumber field
    (adop-put-value (adop-get-item fieldsObj "RoomNumber") "103A")
```

```
      ;; Inform of record change
      (prompt "but is now in room 103A.\n")

      ;; Update the record
      (adom-update adoRecordset)
     (princ)
    )
  )

  ;; Close the recordset and database
  (adom-close adoRecordset)
  (adom-close adoConnection)

  (vlax-release-object adoRecordset)
  (vlax-release-object adoConnection)

  (gc)
 (princ)
)
```

7 Where to Get More Information

When you are first starting to learn a new skill, you will have questions and where you go to find those answers might not be clear. The following is a list of resources that can be helpful:

- **Help System** – The AutoLISP Reference Guide in the AutoCAD online Help system contains information on using some of the COM related functions, but you will also need to turn to the AutoCAD ActiveX Reference Guide.

  To access the online help, go to: https://help.autodesk.com/view/OARX/2019/ENU/
- **Autodesk Discussion Forums** – The Autodesk discussion forums provide peer-to-peer networking that allows you to ask a question about anything related to AutoCAD and get a response from a fellow user or Autodesk employee. To access the Autodesk discussion forums, go to https://forums.autodesk.com, click Browse By Product near the upper-right of the page and then click AutoCAD. Click the appropriate subgroup link.
- **AUGI Forums** – The AUGI forums provide peer-to-peer networking where you can ask questions about virtually anything in AutoCAD and get a response from a fellow user. Visit AUGI at https://www.augi.com/ for more information.
- **Industry Events and Classes** – Industry events such as Autodesk University and BIM Workshops can be great venues to learn about features in an Autodesk product and industry trends. Along with industry events, you might also be able to find classes at a local technical college/university or an Autodesk Authorized Training Center (ATC) that can help you extend your skill set.
- **Internet** – There are samples and tutorials on the Internet that demonstrate other aspects of the AutoCAD and Microsoft Office ActiveX libraries. Use your favorite search engine, such as Google or Bing and search on the topic of interest.
- **Books** – There are many general and specialized books that cover AutoCAD, Windows, and Microsoft Office programming. To find a book, use amazon.com or barnesandnoble.com to locate a book online or visit your local Barnes and Noble store. My latest book, *AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond*, covers all the customization options mentioned in this session and many more.