ES127382

AutoLISP Strategies for CAD Managers

Robert Green
Robert Green Consulting

Learning Objectives

- Learn how to control AutoCAD command vocabularies
- Learn how to craft smarter, more error-tolerant AutoLISP functions
- Learn how to control profile registry variable at startup
- Learn how to compile, secure, and standardize the AutoCAD environment

Description

You can use AutoLISP programming language to very effectively manage AutoCAD-software-based products, although the way to do so isn't always obvious. The trick lies in using AutoLISP to control the command vocabulary while eliminating the need for cumbersome profile-based control of plotting, palettes, and device configuration pathing. We'll discuss how to create, delete, and redefine commands; load external executables; load CUIx and workspaces; control system and registry variables; and make your custom functions better behaved by using ERROR functions and declaring a global variable list for network-wide control. Along the way, we'll explore useful VL/VLR functions and code security. We'll present a sample network environment for context, and provide sample files for download after the class. If you've ever wanted to know more about how AutoLISP can help you manage your AutoCAD software tools, you don't want to miss this class.

Speaker

Since 1991, Robert Green has provided CAD management consulting, programming, training, and technical-writing services for clients throughout the United States, Canada, and Europe. A mechanical engineer by training, Green has used many popular CAD tools in a variety of engineering environments since 1985. Green has acquired his expertise in CAD management via real-world experience as the “alpha CAD user” everywhere he has worked. Over time, he has come to enjoy the technological and training challenges associated with CAD management, and he now trains CAD managers via public speaking. Green is well known for his insightful articles in Cadalyst magazine, and for his book, Expert CAD Management: The Complete Guide (published by Sybex). When he’s not writing, Green heads his own consulting practice, Robert Green Consulting, based in Atlanta, Georgia.

Email: RGreen@GreenConsulting.com

My Philosophy Regarding AutoLISP

AutoLISP is a very powerful language that can take years to learn and master. I've been working with it since 1990 and I still learn new things about AutoLISP on every project. Having said this I'd like to say that there is a lot you can do with AutoLISP without having to be a programming jock if you approach things with a cookbook style approach and use existing routines.

Of particular importance to me, given my CAD manager emphasis, is the ability to really standardize the way AutoCAD behaves, load external commands and centralize all of this from a standardized network location.

I realize there is a wide range of AutoLISP skills in this class so I've tried to build our examples starting with more simple concepts and ratchet the complexity of the code up as we move along.

Key Files and Variables

The first thing to understand is that AutoLISP has a couple of key files and a key function that perform startup operations for you. The key files are called ACAD.LSP and ACADDOC.LSP and the key function is called S::STARTUP and their operations are as summarized here:

ACADDOC.LSP This file loads every time a new drawing session is started in AutoCAD 2xxx based products. Therefore any programming you place in this file will be loaded automatically every time a drawing is opened or started no matter how the ACADLSPASDOC variable is set. Like the ACAD.LSP file, ACADDOC.LSP is normally located in the SUPPORT subdirectory of the AutoCAD installation.

ACAD.LSP Essentially the same as the ACADDOC.LSP but only loads in as AutoCAD loads (not with every new drawing) by default. The system variable ACADLSPASDOC controls this file's loading behavior but, in practicality, there's not advantage using this file over the ACADDOC.LSP file.

MENUNAME.MNL This file loads whenever its menu counterpart (either an MNU, MNC or CUI file) is loaded into AutoCAD. By placing AutoLISP code in this file you can be assured that the code will be loaded into memory ONLY when the parent menu is in use. Note that the code is only loaded once (like the ACAD.LSP) rather than with each drawing session (like the ACADDOC.LSP file). Plan accordingly.

Key Functions

The first thing to understand is that AutoLISP has a couple of key files and a key function that perform startup operations for you. The key files are called ACAD.LSP and ACADDOC.LSP and the key function is called S::STARTUP and their operations are as summarized here:

S::STARTUP function	This function is typically defined in the ACADDOC.LSP file (or ACAD.LSP file for AutoCAD R14 installations) and its sole job is to execute customized commands you need to initialize your new drawing environment. This function is the perfect place to set system variables like DIMSCALE, VIEWRES parameters, current layers, etc. The most powerful aspect of the S::STARTUP function is that it invokes automatically and it lets you control exactly how the AutoCAD environment is initialized.
Reactors	These embedded "reactor" like functions were added when Visual Lisp was bound into AutoCAD 2000. Working with these functions allows HUGE advances in what AutoLISP can do with only minimal overhead. These functions must be invoked (typically from the ACAD.LSP file) by loading the **VL-LOAD-COM** reactors. We'll examine reactors in much more detail a bit later.

Basic ACADDOC.LSP

First let's set some key variables so that each drawing session will get the same basic setup. Just open a text editing program and enter in the following (note the use of ; characters for insertion of comments):

```
(command "viewers" "y" "5000")                ; sets view resolution for no jaggy circles
(command "-color" "BYLAYER")                  ; set color to BYLAYER
(command "-linetype" "set" "BYLAYER" "")      ; set color to BYLAYER
(command "menu" "menuname.mnc")               ; load standard menu file at startup
(prompt "\nACADDOC.LSP loaded … ")            ; send a prompt to the command line
```

Now save this file into your SUPPORT folder with the name ACADDOC.LSP and restart AutoCAD to see if it worked. If everything loaded fine you'll see the prompt appear in your command line window. If not the file was most likely not saved in the correct path.

Q. Why the "." in front of the commands?

A. It has to do with command redefinition which we'll look at shortly.

Q. Why the "-" in front of some commands?

A. It allows you to interact with the command via the command line instead of a dialog box.

Note: Only some commands work with the "-" prefix like -layer, -color, -linetype and –plot which are commonly accessed parameters in many AutoLISP programs.

Conclusion: Obviously you could do much more but even these simple ideas allow you to gain a lot of control with minimal AutoLISP knowledge. In fact, you can actually control user parameters that would otherwise need to be implemented in profiles like menu loading!

Defining Functions (Adding)

You've probably all dabbled with setting up keystroke shortcuts for AutoCAD commands either by editing the ACAD.PGP file directly or by using the Tools -> Customize menu in AutoCAD's interface. This customization methodology allows you to trigger a command using a single keystroke like this:

F triggers the FILLET command.

But what about more complex keystroke combinations that can't be replicated with a single command, like these:

ZOOM ALL

Here's another favorite of mine, the ZA function that peforms a zoom to extents and then zooms out by a factor of 0.95 to create a slight margin around the outside of engineering title blocks:

```
(defun c:za ()
  (command "zoom" "e" "zoom" "0.95x")
  (princ)
)
```

AUTO PURGE

Here's a function that does an autopurge and save by keying in ATP:

```
 (defun c:atp ()
  (command "-purge" "a"  "*" "n" "qsave")
  (princ)
)
```

Back to FILLET

The problem with the single keystroke F is for FILLET approach is that the FILLET command will always use the last setting for the FILLETRAD variable. What if I want to trigger a zero radius fillet? Let's use an AutoLISP function like this:

```
(defun c:fz ()                          ; Define an FZ command
  (setvar “filletrad” 0.0)              ; Set the fillet radius
  (command “fillet” pause pause)        ; Invoke FILLET and wait for two user inputs
  (princ)                               ; Clear the command line
)                                       ; Close the function
```

Make it even better like this:

```
(defun c:fz ()
  (setq old_filletrad (getvar “filletrad”))     ; Store the old fillet radius value in a variable
  (setvar “filletrad” 0.0)
  (command “fillet” pause pause)
  (setvar “filletrad” old_filletrad)            ; Put the fillet radius back
  (princ)
)
```

This example illustrates that simple AutoLISP routines can automate multikeystroke functions with ease!

Note: If you're unsure how to build a function just go to AutoCAD and type in your commands taking careful notes of what you typed. Then just use the syntax I've given here and substitute your values in. Just watch the typos and you'll be writing cool functions very quickly.

Triggering External Commands

What if you wanted to type in NP and have a NotePad session pop up? Do this:

```
(defun c:np ()                          ; Define an NP command
  (startapp “notepad.exe”)              ; Call NotePad
  (princ)                               ; Clear the command line
)                                       ; Close the function
```

So long as you know the name of the EXE and the system has the EXE registered file it'll start.

Note: You can include a path to the EXE file if is not a system registered utility.

Undefining Commands (Removing)

You can undefine a command like this:

```
(command ".undefine" "EXPLODE")
```

Or:

```
(command ".undefine" "LINE")
```

Simply put, you can use this technique to remove troublesome commands from AutoCAD's command set at system startup. So rather than telling people "don't explode your dimensions" you can simply turn off the EXPLODE command!

You can undefine a command like this:

Conclusion: This is a brute force approach to enforcing command usage but it really does work.

Redefining Commands (Overrides)

Now take it another step and combine undefining with a function definition to make a command work the way we want it to work like this:

```
(command ".undefine" "QSAVE")

(defun C:QSAVE ()
  (command "-purge" "b"  "*" "n" ".qsave")
  (princ)
)
```

So what have we accomplished? We've redefined the QSAVE command such that it automatically purges any unused blocks every time the user saves? Tired of telling people to purge the old blocks from their drawings? Problem solved.

What would this code do:

```
(command ".undefine" "LINE")

(defun C:LINE ()
  (command ".pline")
  (princ)
)
```

This example would make the LINE command actually execute PLINE!

Add an Error Handler to FZ

We can see from the FZ command is that if the user hit ESC just prior to the FILLET line that the FILLETRAD system variable would never get reset.

```
(defun c:fz ()
 (setq old_filletrad (getvar “filletrad”))     ; Store the old fillet radius value in a variable
 (setvar “filletrad” 0.0)
 (command “fillet” pause pause)
 (setvar “filletrad” old_filletrad)           ; Put the fillet radius back
 (princ)
)
```

To get around this problem lets create an error handling routine that’ll set FILLETRAD back to the prior value like this:

```
(defun *fz_error* (msg)
 ;; If an error (such as CTRL-C) occurs
 ;; while this command is active...
 (if (/= msg "Function cancelled")
  (princ (strcat "\nError: " msg "\n"))
  (princ)
 )

 (if old_filletrad (setvar " filletrad " old_ fil
 (if *old_error*  (setq *error* *old_error*),

 (princ)
)
```

Note that the old error handler will also be restored so long as you took care to store it in your routine. Your FZ code should now look like this:

```
(defun c:fz ()
 (setq old_filletrad (getvar “filletrad”))        ; Store the old fillet radius value in a variable
 (setq *old_error* *error*)                       ; Store the old error handler
 (setq *error* *fz_error*)
 (setvar “filletrad” 0.0)
 (command “fillet” pause pause)
 (setvar “filletrad” old_filletrad)               ; Put the fillet radius back
 (setq *error* *old_error*)                       ; Put the error handler back
 (princ)
)
```

The key thing to remember is that the error handler in memory is always called as *error* so all you have to do is SETQ *ERROR* to your custom error handler.

Network Support

What if we wanted to store our AutoLISP routines in a single network location so that EVERYBODY had the same AutoLISP files? This approach would allow us to only maintain a single repository of AutoLISP routines and would mean that keeping all machines in sync would only require a single maintenance activity on our part; updating only a single AutoLISP file on the network. I've used this network support approach with many clients and it has not only worked but it has allowed me to keep machines on wide area networks standardized without having to travel or run my feet to a nub in campus style environments.

Here's the basic idea of how to gain control:

- Seed the user's machine with an ACAD.LSP and/or ACADDOC.LSP file in their support directory that will point to an external network location using a variable.
- Execute all load instructions from the ACAD.LSP and/or ACADDOC.LSP using the variable location.
- Test for the presence of all files you'll load using the variable location prior to loading to assure that errors are avoided.

Example ACADDOC.LSP

In this case we wish to load some external AutoLISP files (called UTILS1.LSP and UTILS2.LSP) we've written to add commands to AutoCAD's vocabulary. The case where commands are being added suggests using ACAD.LSP because the commands need only be loaded once, when AutoCAD starts up. Further, we'd like to load these files from a network drive and path that I'll call X:\AUTOLISP as an example.

The contents of the ADADDOC.LSP would look like this:

```
(setq lisp_path “X:\\AUTOLISP\\”)   ; sets the path

(if (findfile (strcat lisp_path "utils1.lsp"))
  (load (strcat lisp_path "utils1.lsp"))
)

(if (findfile (strcat lisp_path "utils2.lsp"))
  (load (strcat lisp_path "utils2.lsp"))
)
```

Notice that the LISP_PATH variable is set first because the following statements make use of the variable to locate the files UTILS1.LSP and UTILS2.LSP in the network drive. This approach gives us the benefits of loading files from a remote location, as opposed to putting all the code in the ACAD.LSP itself, thus we can maintain the remote files separately! On the down side though, we still have to maintain the

ACAD.LSP file on each machine as we include more AutoLISP files for loading. We need a way to get around this problem.

In this case we load in a single remote file from the network location that loads all external functions for us. The new file I'll reference is called INIT.LSP because I use it to INITIALIZE all my file load statements.

The contents of the ADAD.LSP would look like this:

```
(setq lisp_path “X:\\AUTOLISP\\”)   ; sets the path

(if (findfile (strcat lisp_path "init.lsp"))
  (load (strcat lisp_path "init.lsp"))
)
```

In the X:\AUTOLISP\ folder we'll now have a file called INIT.LSP that contains all my SYSVAR settings, function declarations, profile management, etc.

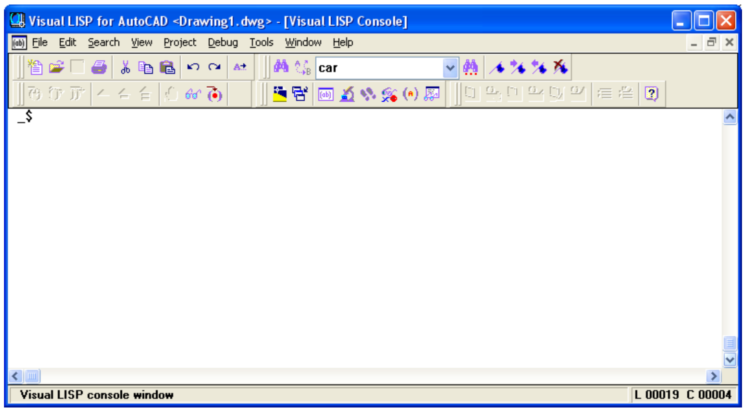
Now we have complete control because the only thing our ACADDOC.LSP file does it set a network location to “look to” and a single initializing file. Now I can make any changes I want in the INIT.LSP file and I'll never have to go to the local machine again unless I need to change the value of the LISP_PATH variable!

This is the way it should be, zero maintenance at the remote machine, total control from the network support path.

Bonus: Note that by setting a variable for the network location in memory that we DO NOT have to make any pathing adjustments in the AutoCAD profile! Again, zero maintenance at the local machine.

Compiling Your Code (VLIDE)

You can also write your code in the VLIDE window (see below) and get the benefit of a drop down function selector and parenthesis matching. Some programmers really like the VLIDE environment while others (myself included) prefer to use their own editing program. Either way is fine. It is simply a matter of preference.


Visual LISP for AutoCAD <Drawing1.dwg> - [Visual LISP Console]
File Edit Search View Project Debug Tools Window Help
car
_$
Visual LISP console window
L 00019 C 00004

One thing I do use the VLIDE interface for is compiling my code into FastLoad (FAS) format which is secure.

Let's say you have your INIT.LSP created and debugged and now you simply want people to run it but never have the ability to edit it. Further, let's say your INIT.LSP file should reside in the X:\AUTOLISP folder. Here's how you'd compile the code:

- Start the VLIDE by typing VLIDE at AutoCAD's command prompt
- In the VLIDE window enter

(vlisp-compile 'st "x:\\autolisp\\init.lsp")

- Hit enter
- You'll now find INIT.FAS in the X:\AUTOLISP folder.

To load this FAS file you'll need to make the following change to your ACADDOC.LSP:

```
(if (findfile (strcat lisp_path "init.fas"))
   (load (strcat lisp_path "init.fas"))
)
```

I really like using the FAS method because I can continue to edit the LSP file while the users only read the compiled FAS file (which is my latest version).

Supporting Power Users (Overrides)

Supporting power users is always a challenge. On the one hand they have the smarts to help themselves so we want to let them. On the other hand, we don't want power users too far outside the standard for fear of introducting errors. Read on for an approach I use to manage non standard users in a standard way.

USER.LSP Architecture

I've found that many companies have power users who have a collection of their own favorite AutoLISP files. The problem becomes one of balancing the need to standardize the way AutoCAD operates without crippling the power users that can do their own AutoLISP work. The solution? Implement an automatic routine that allows users to create their own USER.LSP file that will load in all their AutoLISP code without stepping on the ACAD.LSP or ACADDOC.LSP files you'll use to standardize the installation.

A sample code segment for loading the USER.LSP looks like this:

```
(if (findfile "user.lsp")
  (load "user.lsp")
)
```

The only trick here is that the power user must name his/her file USER.LSP and the file must be located in a path that AutoCAD can find (typically the SUPPORT folder) on the local machine (remember the user is local).

USER.CUIX Architecture

Much like the AutoLISP case we may want to let power users utilize their own, non standard, CUI file. Simple! Use this code segment:

```
(if (findfile "user.cuix")
  (command “menu” "user.cuix")
)
```

Again, the only requirements are the file name and path as in the USER.LSP case.

You may also take the CUIX architecture further by setting a specific workspace from within the CUI file like this:

```
(command “-workspace” ““user”)
```

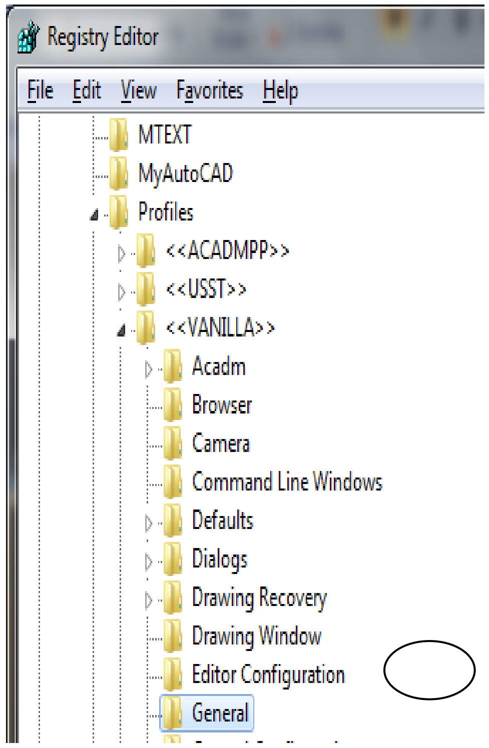
This way the workspace “USER” is made current so long as it exists in the current CUIx

Bypassing Profiles

One of the biggest problems with managing AutoCAD is that you have to work with profiles. Profiles are hard to update and users can always mess with them so no matter how hard you try to manage them you always seem to fall behind.

The solution? Don't worry about profiles, use AutoLISP instead.

The techniques you'll need are registry reads and writes but mainly writes to put the values you want into the user's current profile. This can be achieved SAFELY using AutoLISP **GETENV** and **SETENV** functions like this:



Registry Editor

File Edit View Favorites Help

- MTEXT
- MyAutoCAD
- Profiles
  - <<ACADMPP>>
  - <<USST>>
  - <<VANILLA>>
    - Acadm
    - Browser
    - Camera
    - Command Line Windows
    - Defaults
    - Dialogs
    - Drawing Recovery
    - Drawing Window
    - Editor Configuration
    - General

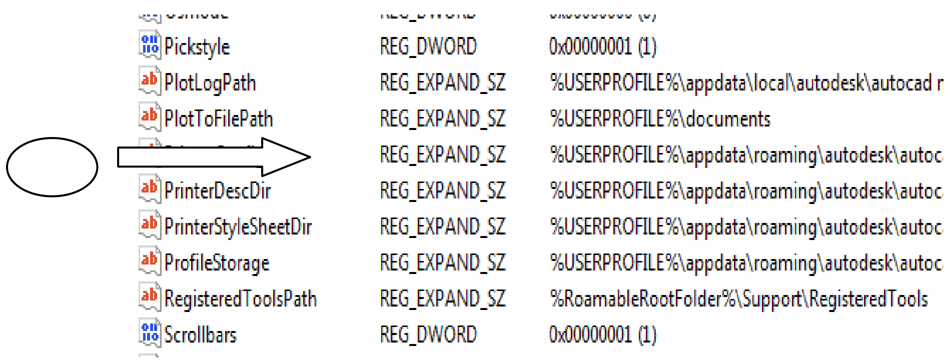
Step 1: Setup a profile that works (I suggest starting with <<VANILLA>>) and then find it using the registry editor application REGEDIT.

Step 2: Next you'll find the specific key that controls the parameter you wish to modify. (In our case it'll be the folder that controls where printer configurations are found.)

Step 3: Note the EXACT spelling and case of the key:

PrinterConfigDir

Note: If you don't get this right nothing will work!



```
Pickstyle            REG_DWORD       0x00000001 (1)
PlotLogPath          REG_EXPAND_SZ   %USERPROFILE%\appdata\local\autodesk\autocad r
PlotToFilePath       REG_EXPAND_SZ   %USERPROFILE%\documents
                     REG_EXPAND_SZ   %USERPROFILE%\appdata\roaming\autodesk\autoc
PrinterDescDir       REG_EXPAND_SZ   %USERPROFILE%\appdata\roaming\autodesk\autoc
PrinterStyleSheetDir REG_EXPAND_SZ   %USERPROFILE%\appdata\roaming\autodesk\autoc
ProfileStorage       REG_EXPAND_SZ   %USERPROFILE%\appdata\roaming\autodesk\autoc
RegisteredToolsPath  REG_EXPAND_SZ   %RoamableRootFolder%\Support\RegisteredTools
Scrollbars           REG_DWORD       0x00000001 (1)
```

Step 4: Override the current profile with an appropriate AutoLISP statement like this:

(setenv "PrinterConfigDir" "Z:\\acad\\Plotters")

or better yet this:

(setenv "PrinterConfigDir" (strcat "Z:\\acad\\Plotters;" (getenv "PrinterConfigDir")))

Let's note a few things:

- In the second example I added my network folder to the existing folder as well
- I took care to put MY network folder first
- I must be sure my network folder exists before adding it to the path
- When I combined paths with STRCAT note the ";" I used to separate the paths
- When the SETENV is called the user's current profile is updated

Expanding on the Idea

With a little investigative work you can discover the following profile variables:

- Support path: **ACAD**
- DRV path: **ACADDRV**
- Printer definitions: **PrinterConfigDir**
- Palettes directory: **ToolPalettePath**
- Templates location: **TemplatePath**
- Plot Styles: **PrinterStyleSheetDir**
- Current CUI: **MenuFile**
- Enterprise CUI: **EnterpriseMenuFile**
- Current Workspace: **WSCURRENT**
- 2014 Trusted Paths: **TRUSTEDPATHS**

It doesn't take too much thinking to realize that you can now totally control the user operating environment no matter what profile the user is in! Of course, the burden is on you to make sure you don't mess anything up – but think of the power you now have to standardize.

Notes on Reactors

In the sample INIT.LSP file you can download you'll see a section on reactors with some sample code in it. This example shows how to detect a user layout switch and set text variables accordingly. It is a great illustration of how to use reactor style controls to standardize a complex user driven task.

```
 (vl-load-com) ; This initializes the reactor environment


(defun DoThisAfterLayoutSwitch (Caller CmdSet)
  (prompt (strcat "\nLayout tab switched to: " (getvar "ctab")))

  (if (= (getvar "userr1") 0.0)
    (setvar "userr1" (getvar "dimscale"))
  )

  (if (= (getvar "ctab") "Model")
    (progn
      (prompt "\nUser MODEL tab switch")
      (setvar "textsize" (* (getvar "dimtxt") (getvar "userr1")))
      (setvar "dimscale" (getvar "userr1"))
      (setvar "ltscale" (* 0.375 (getvar "userr1")))
    )
    (progn
      (prompt "\nUser LAYOUT tab switch")
      (setvar "textsize" (getvar "dimtxt"))
      (setvar "dimscale" 1)
      (setvar "ltscale" 0.375)
    )
  )
  (princ)
)

(setq MyReactor1
  (vlr-miscellaneous-reactor
    nil
    '((:vlr-layoutSwitched . DoThisAfterLayoutSwitch)
     )
  )
)

(defun Clear_My_Reactors ()
  (if (and MyReactor1 (vlr-added-p MyReactor1))
    (vlr-remove MyReactor1)
  )
)

(defun Clear_All_Reactors ( / TMP)
  (vlr-remove-all :vlr-dwg-reactor)
)

(prompt "\nReactors loaded … ")          ; send a diagnostic prompt to the command line
```