November 30 – December 3, 2004 ◊ Las Vegas, Nevada

There's an Error in My Visual LISP® Code. Now What?

R. Robert Bell – MW Consulting Engineers

CP34-2 There are many questions on how to handle errors in Visual LISP. How do I find where errors occur? How do I write an error handler for my application? How do I handle errors when using ActiveX methods and properties? Are there any changes to error handling in recent versions of AutoCAD® software? In this course, we'll ask and answer these questions and more. This session is intended for Visual LISP programmers who want bullet-proof programs.

Who Should Attend
Intermediate-level Visual LISP programmers

Topics Covered
* How to use breakpoints and the Watch window
* What is an error handler?
* Create an error handler to restore user's environment
* How to handle errors while using the ActiveX interface
* Error handling in toolbox functions

About the Speaker:

Robert is the network administrator for MW Consulting Engineers, an engineering consulting firm in Spokane, Washington, where he has worked for the last 16 years. He is responsible for the network and all AutoCAD® customization. Robert has been writing AutoLISP® code since AutoCAD v2.5, and Visual Basic programs for 6 years. He has customized applications for the electrical/lighting, plumbing/piping, and HVAC disciplines. Robert has also developed applications for AutoCAD as a consultant. He is on the Board of Directors for AUGI® and is active on Autodesk newsgroups.

RobertB@MWEngineers.com

The following code will be used throughout the class. This code increments text selected by the user, by using either a numeric or an alphabetic counter. The program permits both prefixed and appended unchanging text, in addition to the counter. Copy the copy into a new file in the Visual LISP® IDE (VLIDE) and save the file (if you wish). Place some text objects, either MText or Text, in the drawing along with some other objects such as Lines. The code has two initial bugs that will be used to demonstrate using a breakpoint and the Watch window.

```
(vl-Load-Com)

(defun I:ReadCounter  (Data)
 (cond ((eq 'INT (type Data)) (itoa Data))
       ((vl-List->String Data))))

(defun I:IncCounter  (Data / Carry)
 (cond ((eq 'INT (type Data)) (1+ Data))
       ((setq Carry (>= (car (setq Data (reverse Data))) 90))
        (setq Data (reverse (mapcar
                             (function (lambda (Alpha)
                                        (cond ((and Carry (> Alpha 90)) 65)
                                              (Carry
                                               (setq Carry nil
                                                     Alpha (1+ Alpha)))
                                              (Alpha))))
                             Data)))
        (cond (Carry (cons 65 Data))
              (Data)))
       ((setq Data (reverse (cons (1+ (car Data)) (cdr Data)))))))

(defun I:InitCounter  (strInp)
 (cond ((> (vl-String-ELT strInp 0) 64) (vl-String->List strInp))
       ((atoi strInp))))

(defun C:Increment  (/ Inc Prefix Suffix Ent)
 (setq Inc    (getstring "\nInitial value <1>: ")
       Inc    (I:InitCounter (strcase (cond ((= Inc "") "1")
                                            (Inc))))
       Prefix (getstring T "\nPrefix with text <>: ")
       Suffix (getstring T "\nAppend with text <>: "))
 (while T
  (while (not (setq Ent (entsel))))
  (cond (Ent
         (vla-Put-TextString
         (vlax-EName->vla-Object Ent)
          (strcat Prefix (I:ReadCounter Inc) Suffix))
         (setq Inc (I:IncCounter Inc)))))
 (princ))
```

Locating where the error occurs

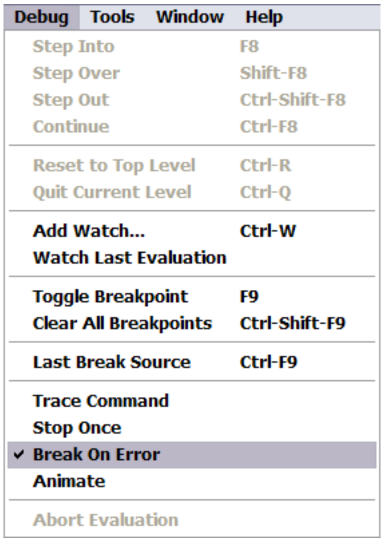
Load the code in the editor window. You can do this by using this button on the Tools toolbar, or the Load Text In Editor item on the Tools menu.

Run the main function (C:Increment) by switching over to AutoCAD and running the newly loaded Increment command. You will get the following error when you pick a text object.

Unselect Break On Error in the Debug menu before running the code the first time.

```
Command: increment
Initial value <1>:
Prefix with text <>:
Append with text <>:
Select object:
Select object: ; error: bad argument type: lentityp (<Entity name: 7ef54e88>
(4.08183 4.70427 0.0))
```

We can tell that we have had an error, but it can be difficult to tell where the error occurred in our code. We see that it is related somehow to an entity, but where is the error?



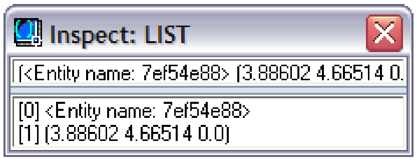
When you are planning to perform a debugging session, you should set the VLIDE to break on any error. You do this by going to the Debug menu and selecting the Break On Error item. This will make it easy to locate the source of the error.

Here is how the process works:

- Run the code
- The VLIDE becomes active when an error occurs
- Use the Last Break button to locate where the error occurred
- The VLIDE highlights the code that caused the error

```
(vla-Put-TextString
 (vlax-EName->vla-Object Ent)
 (strcat Prefix (I:ReadCounter Inc) Suffix))
```

- Inspect the variable Ent to see what it currently is bound to (since it caused the error)

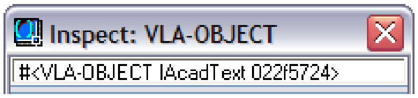


Inspect: LIST

```
(<Entity name: 7ef54e88> (3.88602 4.66514 0.
[0] <Entity name: 7ef54e88>
[1] (3.88602 4.66514 0.0)
```

We can see that the variable is currently bound to a list (the one that comes from the (entsel) function) while the function (vlax-EName->vla-Object) requires an EName only. So a simple change of the code to this will fix this error: (vlax-EName->vla-Object (car Ent))

While the VLIDE is still in break mode, you can verify that the changed code works by re-inspecting the altered statement. This time it should return a vla-Object.



Inspect: VLA-OBJECT

```
#<VLA-OBJECT IAcadText 022f5724>
```

Reset the VLIDE (you may use the Reset button to do so), turn off the Break on Error, and run the code again. You have successfully fixed one error in the program.

Setting a Breakpoint and using the Watch window

Run the Increment command a few more times. Use "1" as the initial value; then run again, using "A" as the initial value. Notice how the counter works with either numbers or alpha characters. The program is intended to increment to "AA", "AB", and so on after "Z". So try starting with an initial value of "Z". Select at least 3 text objects. The first value is "Z", which is correct; the next value is "[", which is incorrect; and the third value is "AA", which is correct. So what is happening? We have a logic error, but this isn't the sort of error that a Break On Error will detect. (In fact, if you turn on Break On Error and run the program, it will not break when the counter goes from "Z" to "[".)

The VLIDE gives us tools to check what is happening while the code is running. Two of these tools are Breakpoints and the Watch window.

We know that the logical error is occurring during the incrementing of the counter. Therefore, we need a way to "break into" the code while it is running in that part of the program. Position your cursor at the beginning of the statement which increments the counter. The statement is: (I:IncCounter Inc).

Set a breakpoint by using the Toggle Breakpoint button or the <F9> key. An active breakpoint is indicated by a red background, i.e. `(setq Inc (I:IncCounter Inc))))`

Run the program again, using “Y” as the initial value (so that we can see a correct increment). Immediately after you pick the 1st text object the VLIDE becomes active, with the statement where the breakpoint is located highlighted.

You may use the <F8> key (my favorite) or the Step Into button to advance the program another statement. You use the <F8> key repeatedly to advance the code statement-by-statement. You may use the Inspect feature at any time to see what are the results of a statement or the value of a variable. When you do this, the highlighted statement might change, but the next time you use the <F8> key it will pick up where it left off. By itself the Breakpoint could be tedious to use. However, when it is teamed with the Watch window, it becomes a powerful tool.

Display the Watch window by using the toolbar button or <Ctrl>+<Shift>+<W>. The Watch window can watch the result of the last expression (*Last-Value*), a variable, or even an expression. Set a watch on the *Last-Value*, the Data variable, and the expression (chr (car Data)) while in the current break. Continue stepping thru the function, watching how the character “Y” is incremented to “Z”.

Once that occurs, you may use the Continue button to exit the current break, which shall allow you to quickly get to the next text object. Manually switch back over to AutoCAD and select another text object. The breakpoint is reached again, so step thru the code to the point where the Data variable is supposed to carry from “Z” to “AA”: (cond ((and Carry (> Alpha 90)) 65). As you reach this test, you can see that the test is incorrect. The statement (> Alpha 90) should be (>= Alpha 90), similar to the test a few lines above. You may use the Reset button to stop execution immediately. Fix the code, reload it, and run it again.

Using an Error Handler

Up to this point we have been concerned about bugs in the program. However, there are other sorts of errors that still affect the usability of the program. For instance, we need to use <Esc> to exit the program, and doing that causes an error:

```
Select object: *Cancel*
; error: Function cancelled
```

This is not exactly a clean way for the program to end. This is something that an error handle can take care of. An error handler is defined using the *Error* function. For instance, the default error handler defined by Visual LISP is this:

```
(defun *error* (msg)
  (princ "error: ")
  (princ msg)
  (princ))
```

It is possible to reset the *Error* function to the equivalent by this statement: (setq *Error* nil).

The *Error* function needs to have one argument, which will be a string returned by Visual LISP when an error occurs. You can define your own error handler for any of your programs, indeed you should. For instance, this version will quietly exit a program when <Esc> is hit, or the functions (exit) or (quit) are used:

```
(defun *Error*  (Msg)
 (cond ((member Msg '("Function cancelled" "quit / exit abort"))) ; <Esc>
       ((princ (strcat "Error: " Msg)) ; display fatal error
        (terpri))) ; print a blank line
 (princ)) ; clean exit
```

When this function is defined externally of other programs, you have created a global error handler. This error handler will run anytime an error occurs in any program or function, unless a local- or application-level error handler is defined. Let us demonstrate this by adding a prompt to the global error handler before the final (princ) statement:

```
(defun *Error*  (Msg)
 (cond ((member Msg '("Function cancelled" "quit / exit abort"))) ; <Esc>
       ((princ (strcat "Error: " Msg)) ; display fatal error
        (terpri)))
 (princ “Running global error handler.”)
 (princ))
```

Load the error handler. Now run the Increment command again. Notice what happens when you hit <Esc>:

```
Select object: *Cancel*
Running global error handler.
```

We can demonstrate further that this error handler is global by simple entering this expression at the command prompt:

```
Command: (/ 1 0)
Error: divide by zero
Running global error handler.
```

There are many times when a global error handler cannot handle all the situations that a specific program might throw at it. In this case, you need to provide…

Local error handlers

A local error handler is simply the *Error* function defined for a specific function. It used to be necessary to save the global error handler, redefine the *Error* function, and use the *Error* function to restore the original error handler. Starting with AutoCAD 2000 this is no longer true. Indeed, AutoCAD’s own documentation has not caught up to this fact.

To make the error handler local, you simply need to declare the symbol as local to the function, and embed the *Error* function in the parent function.

Here is a demonstration of the concept. We will make a simple function that also includes a local error handler.

```
(defun C:Test  (/ *Error* X Y)
;;; Error handler
 (defun *Error*  (Msg)
  (cond ((member Msg '("Function cancelled" "quit / exit abort"))) ; <Esc>
        ((princ (strcat "Error: " Msg)); display fatal error
         (terpri))) ; print a blank line
  (princ "Running local error handler.") ; to show running local handler
  (princ)); clean exit
;;; Main code
 (setq X (getreal "\nThis number: ")
       Y (getreal "\ndivided by this number: "))
 (princ (strcat "\nequals this number: " (rtos (/ x y) 2)))
 (princ))
```

You will get the following when you run this function:

```
Command: test
This number: 12
divided by this number: 0
Error: divide by zero
Running local error handler.
```

You may also verify that that error handler is only local by doing the following:

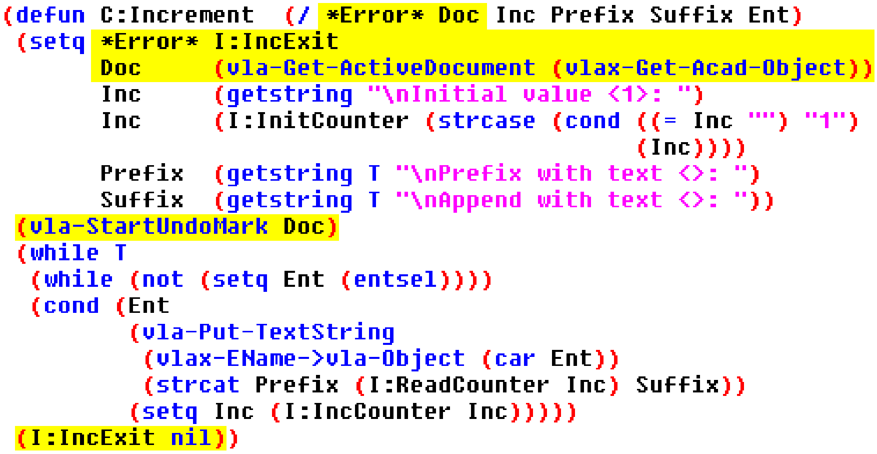
```
Command: (/ 12 0)
Error: divide by zero
Running global error handler.
```

Thus you can see that error handlers can be defined globally or locally. There is also one other level of error handler: the application-level error handler.

Application-level error handlers

It is generally better to design an error handler for an entire application rather than an error handler for each function in the application itself. In this case, you would create a function that is part of the application and just bind the *Error* symbol to that function at the start of your application. This way you will not need to embed the function within another function.

For instance, reset the global error handler by using (setq *Error* nil). Run the Increment command twice, once to modify a few text objects with numbers, then again to modify a few other text objects with letters. Now use the Undo command once. As you can see, both instances of the Increment command were undone. This isn't correct. Each instance of the Increment command should be undone individually.
Not to mention that we still want the <Esc> to exit quietly.



```
(defun C:Increment  (/ *Error* Doc Inc Prefix Suffix Ent)
 (setq *Error* I:IncExit
       Doc     (vla-Get-ActiveDocument (vlax-Get-Acad-Object))
       Inc     (getstring "\nInitial value <1>: ")
       Inc     (I:InitCounter (strcase (cond ((= Inc "") "1")
                                             (Inc))))
       Prefix  (getstring T "\nPrefix with text <>: ")
       Suffix  (getstring T "\nAppend with text <>: "))
 (vla-StartUndoMark Doc)
 (while T
  (while (not (setq Ent (entsel))))
  (cond (Ent
         (vla-Put-TextString
          (vlax-EName->vla-Object (car Ent))
          (strcat Prefix (I:ReadCounter Inc) Suffix))
         (setq Inc (I:IncCounter Inc)))))
 (I:IncExit nil))
```

Let us take the error handlers we have used so far, and make an application-level handler for our application. The first step will be to make some modifications to the main function to provide:

- A localized error handler
- The Active Document
- The start of an Undo Group
- A clean-up function to end the Undo Group, and exit cleanly

As you can see from the modifications to the main function, I will be using a subroutine called I:IncExit to act as both the error handler and the normal exit to the routine. Here is the logic behind that decision. You will find as you become more experienced in writing code that you repeat the same statements to exit a program normally as you do when dealing with an error. The only difference to the error handler would be the code needed to handle the error itself; the cleanup code should be identical to the normal exit. Therefore, if you design the error handler to accept *nil* as an argument, you can make the error handler report no error by passing the function *nil* upon a normal exit. Note the example above: (I:IncExit nil).

Here is the application-level error handler for the Increment program:

```
(defun I:IncExit  (Msg)
 (cond (Doc (vla-EndUndoMark Doc)))
 (cond ((not Msg)) ; normal exit
       ((member Msg '("Function cancelled" "quit / exit abort"))) ; <Esc>
       ((princ (strcat "Error: " Msg)) ; display fatal error
        (terpri) ; print a blank line
        (cond (*Debug* (vl-bt))))) ; if in debug mode, dump backtrace
 (princ))
```

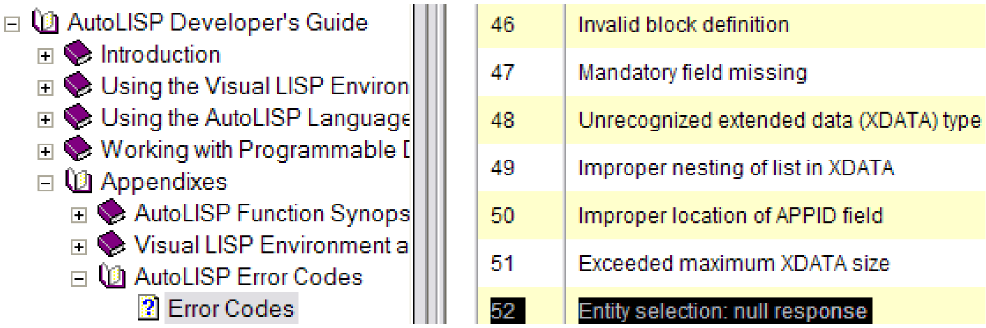
Do you see how the error handler's last statement is a (princ), to provide a clean exit? I am calling this function as the last statement in the main function. I do not need another (princ) in the main function. The same thing applies to the statement (vla-EndUndoMark Doc). Rather than place it in two locations in the application's code, calling the error handler with *nil* as an argument permits me to program it only once. Granted, I am using a test to make sure the Doc variable exists, because I don't want my error handler to **cause** an error if Doc is *nil*.

Run the Increment command twice again. Did you notice how the command-line is cleaner? Now use the Undo command once. Only the last instance of the Increment command is undone. Thus, our application-level error handler has cleaned up the command-line (somewhat) and a potentially nasty bug with undoing.

User interface error handling

To this point the discussion on error handlers has focused on actual "show-stopping" errors. There is another class of errors, related to user interface. Is it really intuitive to the user to have to hit <Esc> to **end** the command? The way the program is currently written causes it to loop forever. The only way to exit the loop is to cause an error by canceling the program. Rather than taking this approach, the program should really handle the "error" of the user hitting <Enter> to exit the program.

Examining the little-known system variable ErrNo can help us accomplish this. ErrNo is set to a value whenever something unusual happens to Visual LISP, even during user interaction. For instance, upon examining the documentation on ErrNo we see that the value 52 is set when there is a null response to an entity selection. In other words, the user hit <Enter> instead of picking an object.


AutoLISP Developer's Guide
Introduction
Using the Visual LISP Environ
Using the AutoLISP Language
Working with Programmable [
Appendixes
AutoLISP Function Synops
Visual LISP Environment a
AutoLISP Error Codes
Error Codes

46	Invalid block definition
47	Mandatory field missing
48	Unrecognized extended data (XDATA) type
49	Improper nesting of list in XDATA
50	Improper location of APPID field
51	Exceeded maximum XDATA size
52	Entity selection: null response

One thing that you need to do before relying on the value of ErrNo is to force it to 0. ErrNo will remain at the last value set, until something happens to change it. Once you have done that, you can examine the value of ErrNo each time thru the loop and quietly exit the loop if the user hits <Enter>.

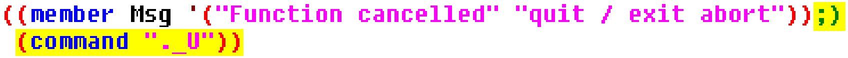
Here are the modifications required to the Increment routine to support what has been discussed. It is actually very simple to provide this functionality:

- Initialize ErrNo
- Replace the "endless" looping caused by (while T ...)
- Remove the unneeded entity selection while loop
- Move the entity selection down to the conditional

```
(defun C:Increment  (/ *Error* Doc Inc Prefix Suffix Ent)
 (setq *Error* I:IncExit
       Doc      (vla-Get-ActiveDocument (vlax-Get-Acad-Object))
       Inc      (getstring "\nInitial value <1>: ")
       Inc      (I:InitCounter (strcase (cond ((= Inc "") "1")
                                              (Inc))))
       Prefix   (getstring T "\nPrefix with text <>: ")
       Suffix   (getstring T "\nAppend with text <>: "))
 (vla-StartUndoMark Doc)
 (setvar "ErrNo" 0)
 (while (/= 52 (getvar "ErrNo"))
  (cond ((setq Ent (entsel))
         (vla-Put-TextString
          (vlax-EName->vla-Object (car Ent))
          (strcat Prefix (I:ReadCounter Inc) Suffix))
         (setq Inc (I:IncCounter Inc)))))
 (I:IncExit nil))
```

Also, because we are no longer forced to use <Esc> to exit the function, we can use the error handler to automatically undo the current increments if the user does decide to cancel the function. All it takes to provide this ability is the following:

- Add an Undo command to the error handler



```
((member Msg '("Function cancelled" "quit / exit abort"));)
 (command "._U"))
```

Another interface issue is what happens when the user selects an object that is not Text or MText. Try it now by selecting a non-text object. You will get this error:

Select object: Error: ActiveX Server returned the error: unknown name: TextString

The way the program is written now it does not handle this type of error. However, this is easy to accommodate also. You could take two approaches:

- Examine what type of entity the user picked, and simply ignore non-text objects
- Filter the user's input so that they may only pick text objects

The first approach is easy, so I will concentrate on the second method instead. The key to the second method is to use arguments available to the (ssget) function that are rarely used or even undocumented.

Here is the statement that will be used to limit the user to selecting only text objects:

```
(ssget "+.:L:S" '((0 . "TEXT,MTEXT")))
```

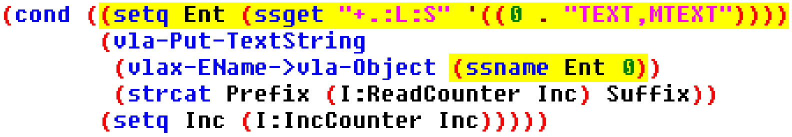
The ":S" argument is documented. It puts the (ssget) function into single selection mode. However, just using that mode by itself is not enough to emulate the (entsel) function. When the user selects empty space a selection window is started.

The undocumented "+." mode forces (ssget) to remain in "point" mode, similar to setting PickAuto to 0. When used in tandem with ":S", this mode creates an effective replacement for (entsel), with the advantage of permitting object filters.

The final mode, ":L", is also undocumented, but very useful. This tells (ssget) to ignore objects on locked layers.

Here are the edits required to provide filtered selection of only text objects for the user:

- Filter the user's selection
- Get the EName of the 1st, and only, object in the selection set



```
(cond ((setq Ent (ssget "+.:L:S" '((0 . "TEXT,MTEXT"))))
       (vla-Put-TextString
        (vlax-EName->vla-Object (ssname Ent 0))
        (strcat Prefix (I:ReadCounter Inc) Suffix))
       (setq Inc (I:IncCounter Inc))))
```

Run the program once again, attempting to select a non-text object. As you can see, it does not take much effort, or code, to handle the interface errors of missing a pick or picking the wrong type of object.

ActiveX error handling

Errors in the ActiveX interface usually require a different approach from using the *Error* function. This is because many ActiveX functions return an **exception** when it cannot perform its operation. Quite often, you even **expect** an exception to occur, simply handle it, and continue the code from the point where the error occurred. This is called "exception-based programming". This form of error handling is common in Visual Basic, but is new territory for the Visual LISP programmer.

An example is in order. The following function accepts a layer name and then sets that layer current (also insuring that the layer is visible and not locked).

```
(defun ax:SLayer  (LayerN / Doc Layer)
 (setq Doc   (vla-Get-ActiveDocument
               (vlax-Get-Acad-Object))
       Layer (vla-Item (vla-Get-Layers Doc) LayerN))
 (vla-Put-Freeze Layer :vlax-False)
 (vla-Put-LayerOn Layer :vlax-True)
 (vla-Put-Lock Layer :vlax-False)
 (vla-Put-ActiveLayer Doc Layer)
 (princ))
```

This routine works great, as long as the layer you are making current is not already current.

```
Command: (getvar "CLayer")
"Test"
Command: (ax:SLayer "0")

Command: (ax:SLayer "0")
; error: Automation Error. Invalid layer
```

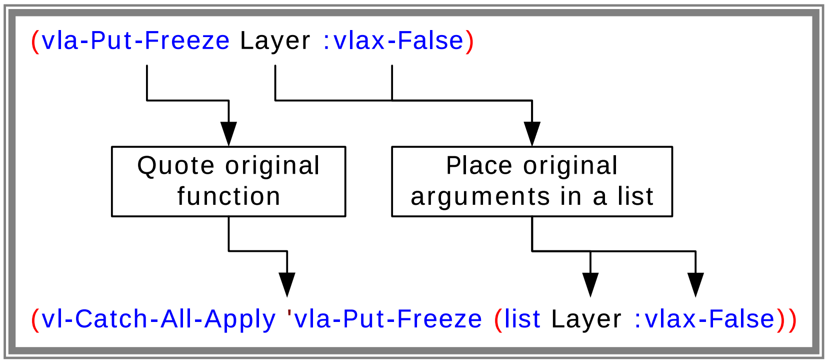
This error occurs because the current layer cannot ever be frozen. So the Freeze property throws an exception when you attempt to write a value to it on the current layer, even if the value is correct (False, that is, not frozen). Note: there is nothing wrong with this code! Therefore, the code needs to handle the exception, and then move on.

The primary function used is (vl-Catch-All-Apply). This function can be confusing if you only look at the example documented in the AutoLISP Reference. Simply put, you need to provide the function to be trapped as a quoted symbol, and the arguments for the trapped function in a list for the second argument. Using the code above, you would modify the problem statement (vla-Put-Freeze Layer :vlax-False) to be this:

```
(vl-Catch-All-Apply 'vla-Put-Freeze (list Layer :vlax-False))
```

Now when you run the routine on the current layer, the exception is handled and the rest of the routine continues. This would have been more difficult with a traditional error handler.

Here is a diagram that details the modification. As you can see, this function is simple to use at this level. You may also use this function to perform some very neat error handling, but that is beyond the scope of this class.



```
(vla-Put-Freeze Layer :vlax-False)
```

```
(vl-Catch-All-Apply 'vla-Put-Freeze (list Layer :vlax-False))
```

There is another function that was introduced in Visual LISP that seems to duplicate an older AutoLISP function: (vl-CmdF). This function is very similar to (command), with one major difference. This function verifies the arguments and if any of them cause an error, it will not execute the command.

An example will make this clear. The following code asks the user to supply 2 points, draws a line between those two points, and inserts a dot at the first point.

```
(defun C:DotLine  (/ Pt1 Pt2)
 (command "._Line"
          (setq Pt1 (getpoint "\nStart point: "))
          (setq Pt2 (getpoint Pt1 "\nEnd point: "))
          "")
 (command "._-Insert" "Dot" Pt1 (getvar "DimScale") "" "")
 (princ))
```

This is the simple way to write this code. However, if the user cancels one of the (getpoint) statements, the code will not cancel the Line command. The only alternative would be to get the points and test to make sure you had valid points before executing the Line command.

Here is the same code using the (vl-CmdF) function. Now, when the user cancels the program, the Line command will not execute.

```
(defun C:DotLine  (/ Pt1 Pt2)
 (vl-CmdF "._Line"
          (setq Pt1 (getpoint "\nStart point: "))
          (setq Pt2 (getpoint Pt1 "\nEnd point: "))
          "")
 (vl-CmdF "._-Insert" "Dot" Pt1 (getvar "DimScale") "" "")
 (princ))
```

It is important to note that this function is not perfect; it cannot trap the error that a missing block would generate, for instance.

Error handling in “Toolbox” functions

After all this discussion on error handlers, the first impulse is to “error-proof” all your routines, including your “toolbox” functions. This is a mistake. Your own toolbox routines should run fast; be lean, mean, coding machines. Some error handling is appropriate, but the code needs to be efficient. If it attempts to cover all mistakes that will be thrown at it, it will run noticeably slower.

You are the programmer. You need to verify your data before you pass it to another function. Your toolbox functions should be able to rely on the arguments passed to them. Consider the function (entsel) for a moment. Is it possible to give it an incorrect argument? Does the function quietly handle the error? Or does it throw an error and a message and leave it at that? Notice that the function intentionally returns an error when you give it invalid data. You will find that many of your own toolbox functions will perform best when they are designed in the same manner.

Another way that you can reduce errors is by creating self-documenting code. This is code that sacrifices elegance for readability and future code maintenance. As you become a more accomplished programmer, you tend to admire elegant code and might find yourself writing code that is so cool that you cannot figure out what you did two years later! So balance elegant, tight code against readability. Years from now you will appreciate simple-to-read code.

It is also important to document your code with comments and headers that explain the operation of your code. Here is a sample of the header that I use for my code:

```
;|

GetInp.lsp

Version history
2.31   2003/05/27   Minor reformatting.
2.3    2003/05/10   Added newline to the inp list string, if strPrompt was nil.
2.2    2001/09/20   Fixed bug that did not allow last condition (getpoint and such) to
                    evaluate correctly. getType variable must be a quoted function.
2.1    2001/08/28   Replace (if) on defaults to a (cond).
2.01   2001/05/18   Modified kWordStr's (setq) to use Michael Puckett's (mapcar)
                    style to make a list into delimited string (instead
                    of using (vl-string-left trim) to remove leading slash).
2.0    2001/05/17   Modified to work with any (get...) function.
                    Takes arguments as list. Default may be either
                    string or dxf style list of (STR . VAL).
1.2    2000/07/13   Now use string for default. Format options in AC2K style.
1.1    1997/01/28   Can use nil for Prompt string.
1.0    1996/08/20   Initial release.

Returns user selection using a list of strings as keywords.

Dependencies:none
Usage:       (rrbI:GetInp Data)
Arguments:   Data         list ([strPrompt] [Default] [InpList] [getType] [okSpaces/Pt]
                          [initKey])
             strPrompt    string, descriptive text that preceeds options, may be nil.
             Default      string or list, default input, may be nil. If list,
                          format as (string . value).
             InpList      list, strings for valid input, may be nil.
             getType      (get...) function name (quoted, e.g. 'getint), for use to get
                          input, may be nil.
             okSpaces/Pt  if T, allow spaces in (getstring), or...
                          to provide optional Pt argument to some (get...) functions.
             initKey      integer, for use with (initget), may be nil.
Returns:     return from (get...) function, or default string.

Copyright © 1996-2003 by R. Robert Bell.
RobertB@acadx.com

|;
```

This header may be more complete than you require, but after many years of writing code, I've found this type of header to be the best for me. It documents the revisions to my code, the purpose of the routine, any other toolbox functions it might be using, its usage and the arguments it expects, and finally, what the function returns.

Conclusion

Error handling can be a challenge, yet you get a great amount of satisfaction from chasing down the bugs and squashing them. This course has introduced you to several techniques that will be useful project-after-project. The approach to debugging, at the beginning of the course, will be of use every time you run into a fatal error in your code. The code to improve the user interface will prove valuable every time you interact with the user.

Please enjoy the rest of Autodesk University! If you have any questions, feel free to contact me.