AFRALISP

The AutoLisp Tutorials

AutoLisp

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

afralisp@mweb.com.na
http://www.afralisp.com

Contents

Page 6	**The Basics in a Nutshell.**
Page 27	**Loading Autolisp Files**
Page 30	**AutoCAD and Customizable Support Files**
Page 41	**Environment Variables Listing**
Page 46	**Migrating Express Tools**
Page 47	**Partial Menu's**
Page 64	**Automating Menu Loading**
Page 66	**Creating Hatch Patterns**
Page 69	**Creating Linetypes**
Page 77	**Menu Loading**
Page 83	**Define Function (defun)**
Page 87	**Program Looping**
Page 90	**Conditionals.**
Page 94	**Error Trapping**
Page 98	**Calculating Points (Polar)**
Page 100	**Locating Files.**
Page 103	**File Handling.**
Page 109	**External Data**
Page 112	**List Manipulation.**
Page 121	**Into the DataBase.**
Page 132	**DXF Group Codes.**
Page 135	**Selection Sets.**
Page 139	**Selection Set Filters**
Page 142	**Working With Layers & Styles.**
Page 144	**Polylines and Blocks.**
Page 149	**Extended Entity Data.**
Page 161	**(mapcar) and (lambda)**
Page 167	**The 'Eval' Function.**
Page 172	**Redefining Commands.**
Page 176	**Efficient Variables.**
Page 180	**The "DIR" Command.**
Page 183	**Colours and Linetypes ByLayer**
Page 189	**Debugging**
Page 195	**Dictionaries and XRecords.**

Page 201	**Drawing Setup**
Page 217	**AutoLisp/VBA Drawing Setup**
Page 229	**Application Data.**
Page 231	**Time and Date Stamping.**
Page 236	**AutoLisp Macro Recorder.**
Page 240	**Auto-Breaking Blocks.**
Page 244	**List Variables/Functions.**
Page 249	**Lisp Help.**
Page 253	**DOSLib - Batch Slides.**
Page 263	**AfraLisp Slide Manager**
Page 265	**Working with Areas**
Page 270	**AutoCAD and HTML - AutoLisp**
Page 273	**AutoCAD and HTML - Revisited**
Page 277	**Environmental Issues**
Page 279	**Quick Plot**
Page 285	**Acknowledgements and Links**

The Basics in a Nutshell

So, you've never programmed in AutoLisp before!
You've tried to decipher some AutoLisp routines but, you are still totally confused!!!
Let's see if we can't help you out.
This tutorial will try and teach you the very basics of AutoLisp programming without overwhelming you with double-gook.

Let's start up with something very simple and that will give you immediate results. Fire up AutoCad and type this at the command prompt:

```
(alert "Yebo Gogo")
```

Now press enter. This should appear on your screen :

Well Done, you've just used AutoLisp to make AutoCad do something.
By the way, most other programming tutorials use "Hello World" as a similar example. But, because this is an African site, I thought that I would use a very well known Zulu phrase instead.

As you noticed using the (alert) function results in a dialogue box being displayed on your screen.

Let's try something else. Type this at the command prompt and press enter :

```
(setq a (getpoint))
```

Then choose a point anywhere on the screen.
A "list" of numbers, looking something like this, should appear in your command window.

```
(496.0 555.06 0.0)
```

This list, believe it or not, contains the x, y and z coordinates of the point you picked.

```
x = 496.04
y = 555.06
z = 0.0
```

The AutoLisp coding :

```
(setq a (getpoint))
```

Means, in plain English :

```
Get a point from the user and store the x, y and z
values as a list in variable "a".
```

Did you notice how everything is enclosed within parenthesis?
All AutoLisp functions are surrounded by parenthesis.
As well, AutoLisp allows you to "nest" your functions.
This lets you write a function that evaluates another function.
Just remember, that you must leave the nest with an equal number of parenthesis. Here's an example :

```
(dosomething (dosomethingelse (andanotherthing)))
```

You could also write the above statement like this to make it more readable :

```
(dosomething
        (dosomethingelse
                (andanotherthing)
        )
)
```

Now you can see why "Lisp" is often known as "Lost in Stupid Parenthesis"

You can also add comments to your coding. Anything preceded with a semicolon is not evaluated by Autolisp and is treat as a comment, much the same way as the REM statement in Basic is used. e.g.

```
(dosomething
        (dosomethingelse
                (andanotherthing)  ;This is a comment
        )  ;This is another comment
)  ;and another comment
```

The statement we wrote earlier, told AutoLisp to get a point from the user and store the value in variable "a".
Now type this at the command line :

```
!a
```

The point list should be returned. So, any time that you would like to inspect a variable, just precede the variable name with "!"

Our getpoint function worked, but it didn't really tell the user what was expected from him by way of input. Try this now :

```
(setq a ( getpoint "\nChoose a Point : "))
```

Did you notice how Autolisp now asks you for input (and what type of input is expected.)

Let's write a programme.
Type in each of these lines, one at a time, pressing "Enter" at the end of each line, then choosing a point.

```
(setq a (getpoint "\nEnter First Point : "))
```

Press "Enter" then select a point.

```
(setq b (getpoint "\nEnter Second Point : "))
```

Again, Press "Enter" then select a second point.

```
(command "Line" a b "")
```

Press "Enter" again. A line should be drawn between the two points.
The (command) function is used to tell AutoCad what you want it to do.

```
"Line"  Draw a Line
a       From the point stored in variable "a"
b       To the point stored in variable "b"
""      Enter to close the Line command.
```

Now this is very nice, but do we have to type in all this coding every time we want to use this routine?

Next we will discuss how to "store" your programs in a file.

We now need to be able to "store" our AutoLisp routines in a file.
AutoLisp files are simple ASCII text files with the extension "lsp".
Open up Notepad or any other simple text editor and type in the following :

```
(defun testline ()
        ;define the function

        (setq a (getpoint "\nEnter First Point : "))
        ;get the first point

        (setq b (getpoint "\nEnter Second Point : "))
        ;get the second point

        (command "Line" a b "")
        ;draw the line

)       ;end defun
```

Now, save this file as "testline.lsp" remembering, to save it as a ASCII Text file and ensuring that it is saved in a directory in your AutoCAD's search path. Now open AutoCAD and type this :

```
        (load "testline")
```

This will load the function into memory. (Did you notice that you do not have to stipulate the "lsp" extension?) Now type this :

```
        (testline)
```

Your function should now run.

Let's edit this routine so that it acts like a standard AutoCAD command :

```
(defun c:testline ()
        ;define the function

        (setq a (getpoint "\nEnter First Point : "))
        ;get the first point

        (setq b (getpoint "\nEnter Second Point : "))
        ;get the second point

        (command "Line" a b "")
        ;draw the line

)       ;end defun
```

By preceding the function name with c: we do not have to enclose the name with brackets when running. Re-load the function and run it again.

```
        (load "testline")
        testline
```

Much better, Hey.
We do have one problem though. Did you notice that when we loaded the routine, an annoying "nil" kept on popping up. We also got the same "nil" returned to us when we ran the routine. To suppress this "nil" when loading or running, we can use the (princ) function. Here's what your routine would look like :

```
(defun c:testline ()
        ;define the function

        (setq a (getpoint "\nEnter First Point : "))
        ;get the first point

        (setq b (getpoint "\nEnter Second Point : "))
        ;get the second point

        (command "Line" a b "")
        ;draw the line

        (princ)
        ;clean running

)       ;end defun
(princ)
;clean loading
```

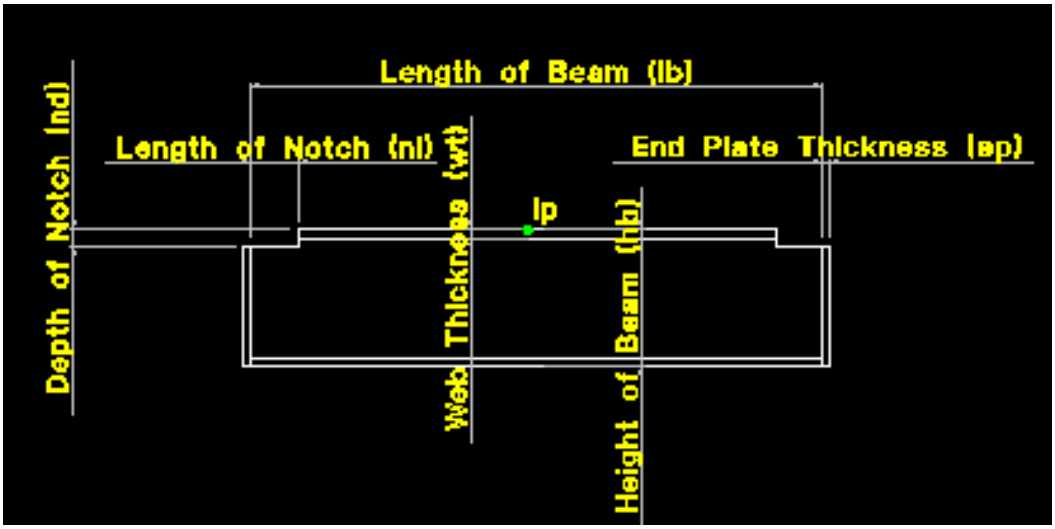
For more details on the (defun) function, refer to The AfraLisp Tutorial :

Define Function (defun).

And for a more detailed expanation of loading AutoLisp routines, refer to the AfraLisp tutorial :

Loading AutoLisp Files.

Now it's time to make AutoLisp do some calculations for us.
Let's say we wanted AutoLisp to draw a beam, in elevation, for us.
First of all we would start by getting input from the user regarding certain parameters that we would need to draw the beam.
Here's what we are trying to draw, along with the values that the user needs to input.



The values that we need to retrieve from the user are as follows :

```
Insertion Point          ip
Length of Beam           lb
Height of Beam           hb
Flange Thickness         wt
End Plate Thickness      ep
Length of Notch          nl
Depth of Notch           nd
```

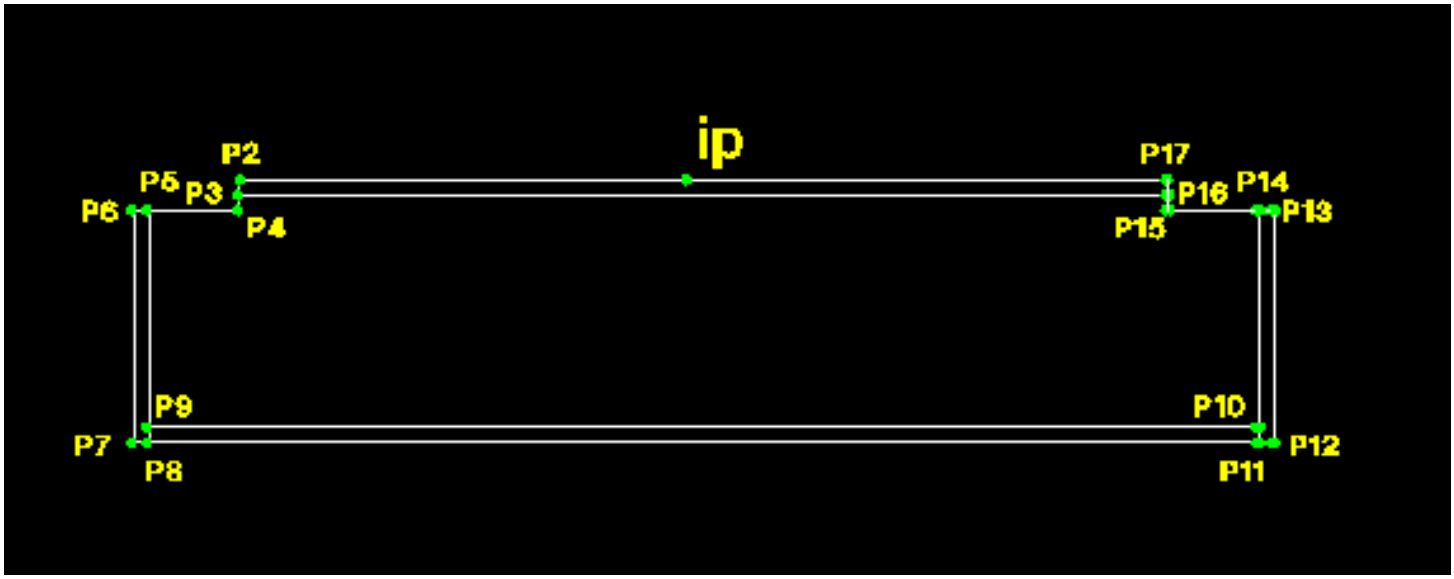
Let's write a routine to retrieve these values first.

```
(defun c:testbeam ()
        ;define the function

;*************************************************************
        ;Get User Inputs

        (setq lb (getdist "\nLength of Beam : "))
        ;get the length of the beam

        (setq hb (getdist "\nHeight of Beam : "))
        ;get the height of the beam

        (setq wt (getdist "\nFlange Thickness : "))
        ;get the thickness of the flange

        (setq ep (getdist "\nEnd Plate Thickness : "))
        ;get the thickness of the end plate

        (setq nl (getdist "\nLength of Notch : "))
        ;get the length of notch

        (setq nd (getdist "\nDepth of Notch : "))
        ;get the depth of the notch
```

```
        ;End of User Inputs
;*************************************************************
        ;Get Insertion Point

        (setq ip (getpoint "\nInsertion Point : "))
        ;get the insertion point

;*************************************************************
        (princ)
        ;finish cleanly

)       ;end of defun

;*************************************************************
(princ) ;load cleanly
;*************************************************************
```

Load and run the routine. You will be prompted for all the values listed above. Enter some numbers and then check the value of all the variables by preceeding the value names with "!" (e.g. !ip).

O.K. we've got all the values we need from the user.
But first we need to do some calculations to determine the other points required before we can draw the beam.



Well, as you can see, we have quite a few points that we need to calculate.
Fortunately, AutoLisp has a function to help us, the (polar) function.
The (polar) function works like this :

You pass it a point, an angle and a distance and (polar) will return a second point at the specified angle and distance from the first point.

But, we have one problem. All angles in AutoLisp MUST be given in radians.
Let's quickly write a function to do that. Add this function to your Testbeam.lsp file :

```
(defun dtr (x)
        ;define degrees to radians function

        (* pi (/ x 180.0))
        ;divide the angle by 180 then
        ;multiply the result by the constant PI

)       ;end of function
```

O.K. thats our "degrees to radians" function taken care of.
Now we'll have a look at the (polar) function in action.

```
        (setq p2 (polar ip (dtr 180.0) (- (/ lb 2) nl)))
```

What we are saying here is :

"Set the variable "p2" to a point that, from the insertion point "ip", at an angle of 180.0 degrees (converted to radians), is at a distance of the length of the beam divided by 2, minus the length of the notch."

This will calculate and return the point "p2". Let's do the rest :

```
(defun c:testbeam ()
```

```
        ;define the function

;*************************************************************
        ;Get User Inputs

        (setq lb (getdist "\nLength of Beam : "))
        ;get the length of the beam

        (setq hb (getdist "\nHeight of Beam : "))
        ;get the height of the beam

        (setq wt (getdist "\nFlange Thickness : "))
        ;get the thickness of the flange

        (setq ep (getdist "\nEnd Plate Thickness : "))
        ;get the thickness of the end plate

        (setq nl (getdist "\nLength of Notch : "))
        ;get the length of notch

        (setq nd (getdist "\nDepth of Notch : "))
        ;get the depth of the notch

        ;End of User Inputs
;*************************************************************
        ;Get Insertion Point

        (setq ip (getpoint "\nInsertion Point : "))
        ;get the insertion point

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

        ;Start of Polar Calculations

        (setq p2  (polar ip (dtr 180.0) (- (/ lb 2) nl)))
        (setq p3  (polar p2 (dtr 270.0) wt))
        (setq p4  (polar p2 (dtr 270.0) nd))
        (setq p5  (polar p4 (dtr 180.0) nl))
        (setq p6  (polar p5 (dtr 180.0) ep))
        (setq p7  (polar p6 (dtr 270.0) (- hb nd)))
        (setq p8  (polar p7 (dtr 0.0) ep))
        (setq p9  (polar p8 (dtr 90.0) wt))
        (setq p10 (polar p9 (dtr 0.0) lb))
        (setq p11 (polar p8 (dtr 0.0) lb))
        (setq p12 (polar p11 (dtr 0.0) ep))
        (setq p13 (polar p12 (dtr 90.0) (- hb nd)))
        (setq p14 (polar p13 (dtr 180.0) ep))
        (setq p15 (polar p14 (dtr 180.0) nl))
        (setq p16 (polar p15 (dtr 90.0) (- nd wt)))
        (setq p17 (polar p16 (dtr 90.0) wt))
        ;End of Polar Calculations
```

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

        (princ)
        ;finish cleanly

)       ;end of defun

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

;This function converts Degrees to Radians.

(defun dtr (x)
        ;define degrees to radians function

        (* pi (/ x 180.0))
        ;divide the angle by 180 then
        ;multiply the result by the constant PI

)       ;end of function

;*************************************************************
(princ) ;load cleanly
;*************************************************************
```

Right, now that we've calculated all the points required to draw the beam, let us add a (command) function to do this task for us.

```
(defun c:testbeam ()
        ;define the function
;*************************************************************

        ;Get User Inputs

        (setq lb (getdist "\nLength of Beam : "))
        ;get the length of the beam

        (setq hb (getdist "\nHeight of Beam : "))
        ;get the height of the beam

        (setq wt (getdist "\nFlange Thickness : "))
        ;get the thickness of the flange

        (setq ep (getdist "\nEnd Plate Thickness : "))
        ;get the thickness of the end plate

        (setq nl (getdist "\nLength of Notch : "))
        ;get the length of notch
```

```
        (setq nd (getdist "\nDepth of Notch : "))
        ;get the depth of the notch

        ;End of User Inputs
;*************************************************************
        ;Get Insertion Point

        (setq ip (getpoint "\nInsertion Point : "))
        ;get the insertion point

;*************************************************************
        ;Start of Polar Calculations

        (setq p2  (polar ip (dtr 180.0) (- (/ lb 2) nl)))
        (setq p3  (polar p2 (dtr 270.0) wt))
        (setq p4  (polar p2 (dtr 270.0) nd))
        (setq p5  (polar p4 (dtr 180.0) nl))
        (setq p6  (polar p5 (dtr 180.0) ep))
        (setq p7  (polar p6 (dtr 270.0) (- hb nd)))
        (setq p8  (polar p7 (dtr 0.0) ep))
        (setq p9  (polar p8 (dtr 90.0) wt))
        (setq p10 (polar p9 (dtr 0.0) lb))
        (setq p11 (polar p8 (dtr 0.0) lb))
        (setq p12 (polar p11 (dtr 0.0) ep))
        (setq p13 (polar p12 (dtr 90.0) (- hb nd)))
        (setq p14 (polar p13 (dtr 180.0) ep))
        (setq p15 (polar p14 (dtr 180.0) nl))
        (setq p16 (polar p15 (dtr 90.0) (- nd wt)))
        (setq p17 (polar p16 (dtr 90.0) wt))
        ;End of Polar Calculations
;*************************************************************

        ;Start of Command Function

        (command "Line" ip p2 p4 p6 p7 p12 p13 p15 p17 "c"
                 "Line" p3 p16 ""
                 "Line" p9 p10 ""
                 "Line" p5 p8 ""
                 "Line" p11 p14 ""
        )        ;End Command
        ;End of Command Function

;*************************************************************
        (princ)
        ;finish cleanly

)       ;end of defun

;*************************************************************
;This function converts Degrees to Radians.
```

```
(defun dtr (x)
        ;define degrees to radians function

        (* pi (/ x 180.0))
        ;divide the angle by 180 then
        ;multiply the result by the constant PI

)       ;end of function

;*************************************************************
(princ) ;load cleanly
;*************************************************************
```

O.K. Let's load the program and run it.
Now, depending on how your snap is set, the beam might be drawn correctly, or it may not. On my system it doesn't draw properly.
This is because my snap defaults to intersection and AutoCad keeps on snapping to the wrong point. I also have lot's of annoying "blips" on my screen. Now we'll discuss how to get around these problems.

Snap settings in AutoCad can cause havoc with an AutoLisp routine.
Therefore, it's a very good idea to switch off the snap at the beginning of any AutoLisp routine and only switch it on when it is required. Blips on your screen are also very annoying, so switch them off too. Please, just remember to return the AutoCad settings that you change, back to their original state before exiting your routine.
Let's edit our routine to include this :

```
(defun c:testbeam ()
        ;define the function
;*******************************************************

        ;Save System Variables

        (setq oldsnap (getvar "osmode"))
        ;save snap settings

        (setq oldblipmode (getvar "blipmode"))
        ;save blipmode setting

;*******************************************************
        ;Switch OFF System Variables

        (setvar "osmode" 0)
        ;Switch OFF snap

        (setvar "blipmode" 0)
        ;Switch OFF Blipmode

;*******************************************************
        ;Get User Inputs

        (setq lb (getdist "\nLength of Beam : "))
        ;get the length of the beam

        (setq hb (getdist "\nHeight of Beam : "))
        ;get the height of the beam

        (setq wt (getdist "\nFlange Thickness : "))
        ;get the thickness of the flange

        (setq ep (getdist "\nEnd Plate Thickness : "))
        ;get the thickness of the end plate

        (setq nl (getdist "\nLength of Notch : "))
        ;get the length of notch

        (setq nd (getdist "\nDepth of Notch : "))
        ;get the depth of the notch

        ;End of User Inputs
```

```
;*************************************************************
        ;Get Insertion Point

        (setvar "osmode" 32)
        ;switch ON snap

        (setq ip (getpoint "\nInsertion Point : "))
        ;get the insertion point

        (setvar "osmode" 0)
        ;switch OFF snap

;*************************************************************
        ;Start of Polar Calculations

        (setq p2  (polar ip (dtr 180.0) (- (/ lb 2) nl)))
        (setq p3  (polar p2 (dtr 270.0) wt))
        (setq p4  (polar p2 (dtr 270.0) nd))
        (setq p5  (polar p4 (dtr 180.0) nl))
        (setq p6  (polar p5 (dtr 180.0) ep))
        (setq p7  (polar p6 (dtr 270.0) (- hb nd)))
        (setq p8  (polar p7 (dtr 0.0) ep))
        (setq p9  (polar p8 (dtr 90.0) wt))
        (setq p10 (polar p9 (dtr 0.0) lb))
        (setq p11 (polar p8 (dtr 0.0) lb))
        (setq p12 (polar p11 (dtr 0.0) ep))
        (setq p13 (polar p12 (dtr 90.0) (- hb nd)))
        (setq p14 (polar p13 (dtr 180.0) ep))
        (setq p15 (polar p14 (dtr 180.0) nl))
        (setq p16 (polar p15 (dtr 90.0) (- nd wt)))
        (setq p17 (polar p16 (dtr 90.0) wt))
        ;End of Polar Calculations
;*************************************************************
        ;Start of Command Function

        (command "Line" ip p2 p4 p6 p7 p12 p13 p15 p17 "c"
                 "Line" p3 p16 ""
                 "Line" p9 p10 ""
                 "Line" p5 p8 ""
                 "Line" p11 p14 ""
        )        ;End Command
        ;End of Command Function
;*************************************************************

        ;Reset System Variable

        (setvar "osmode" oldsnap)
        ;Reset snap

        (setvar "blipmode" oldblipmode)
```

```
        ;Reset blipmode

;*************************************************************
        (princ)
        ;finish cleanly

)       ;end of defun

;*************************************************************
;This function converts Degrees to Radians.

(defun dtr (x)
        ;define degrees to radians function

        (* pi (/ x 180.0))
        ;divide the angle by 180 then
        ;multiply the result by the constant PI

)       ;end of function

;*************************************************************
(princ) ;load cleanly
;*************************************************************
```

Load and run the program. If you were having problems, they should now have disappeared. (I hope!!). Did you notice how we switched on the snap just before asking for the insertion point?
This, of course, is to allow the user to snap to an insertion point.

We still have another problem though. What would happen if the user entered an illegal value, such as zero, or a negative number.
This could cause very strange results.

To guard against this we will use the (initget) function.
Here's how it works :

```
        (initget (+ 1 2 4))
        (setq lb (getdist "\nLength of Beam : "))
```

This function works on the "sum of the bits" system.

```
        1 = Disallows the user from pressing "Enter".
        2 = Disallows the user from entering "Zero".
        3 = Disallows the user from entering a "Negative Number".
```

The same function could have been written like this :

```
        (initget 7)
```

```
        (setq lb (getdist "\nLength of Beam : "))
```

Let's add this function to our routine :

```
(defun c:testbeam ()
        ;define the function
;*******************************************************
        ;Save System Variables

        (setq oldsnap (getvar "osmode"))
        ;save snap settings

        (setq oldblipmode (getvar "blipmode"))
        ;save blipmode setting

;*******************************************************
        ;Switch OFF System Variables

        (setvar "osmode" 0)
        ;Switch OFF snap

        (setvar "blipmode" 0)
        ;Switch OFF Blipmode

;*******************************************************
        ;Get User Inputs

        (initget (+ 1 2 4))
        ;check user input

        (setq lb (getdist "\nLength of Beam : "))
        ;get the length of the beam

        (initget (+ 1 2 4))
        ;check user input

        (setq hb (getdist "\nHeight of Beam : "))
        ;get the height of the beam

        (initget (+ 1 2 4))
        ;check user input

        (setq wt (getdist "\nFlange Thickness : "))
        ;get the thickness of the flange

        (initget (+ 1 2 4))
        ;check user input

        (setq ep (getdist "\nEnd Plate Thickness : "))
        ;get the thickness of the end plate
```

```
        (initget (+ 1 2 4))
        ;check user input

        (setq nl (getdist "\nLength of Notch : "))
        ;get the length of notch

        (initget (+ 1 2 4))
        ;check user input

        (setq nd (getdist "\nDepth of Notch : "))
        ;get the depth of the notch

        ;End of User Inputs
;*************************************************************
        ;Get Insertion Point

        (setvar "osmode" 32)
        ;switch ON snap

        (setq ip (getpoint "\nInsertion Point : "))
        ;get the insertion point

        (setvar "osmode" 0)
        ;switch OFF snap

;*************************************************************
        ;Start of Polar Calculations

        (setq p2  (polar ip (dtr 180.0) (- (/ lb 2) nl)))
        (setq p3  (polar p2 (dtr 270.0) wt))
        (setq p4  (polar p2 (dtr 270.0) nd))
        (setq p5  (polar p4 (dtr 180.0) nl))
        (setq p6  (polar p5 (dtr 180.0) ep))
        (setq p7  (polar p6 (dtr 270.0) (- hb nd)))
        (setq p8  (polar p7 (dtr 0.0) ep))
        (setq p9  (polar p8 (dtr 90.0) wt))
        (setq p10 (polar p9 (dtr 0.0) lb))
        (setq p11 (polar p8 (dtr 0.0) lb))
        (setq p12 (polar p11 (dtr 0.0) ep))
        (setq p13 (polar p12 (dtr 90.0) (- hb nd)))
        (setq p14 (polar p13 (dtr 180.0) ep))
        (setq p15 (polar p14 (dtr 180.0) nl))
        (setq p16 (polar p15 (dtr 90.0) (- nd wt)))
        (setq p17 (polar p16 (dtr 90.0) wt))
        ;End of Polar Calculations
;*************************************************************
        ;Start of Command Function

        (command "Line" ip p2 p4 p6 p7 p12 p13 p15 p17 "c"
                 "Line" p3 p16 ""
```

```
                "Line" p9 p10 ""
                "Line" p5 p8 ""
                "Line" p11 p14 ""
        )       ;End Command
        ;End of Command Function
;*******************************************************
        ;Reset System Variable

        (setvar "osmode" oldsnap)
        ;Reset snap

        (setvar "blipmode" oldblipmode)
        ;Reset blipmode

;*******************************************************
        (princ)
        ;finish cleanly

)       ;end of defun

;*******************************************************
;This function converts Degrees to Radians.

(defun dtr (x)
        ;define degrees to radians function

        (* pi (/ x 180.0))
        ;divide the angle by 180 then
        ;multiply the result by the constant PI

)       ;end of function

;*******************************************************
(princ) ;load cleanly
;*******************************************************
```

Now, everything should be running fine except for one thing.
What happens if we want to draw two beams with exactly the same values? We would have to go through the whole routine again, entering exactly the same inputs. We could, of course, set up each variable to default to the last value used, but we would still have to run through the whole routine. Here's a better way. We'll enclose the complete routine in a (while) loop. Have a look :

```
(defun c:testbeam ()
        ;define the function
;*******************************************************
        ;Save System Variables
```

```
        (setq oldsnap (getvar "osmode"))
        ;save snap settings

        (setq oldblipmode (getvar "blipmode"))
        ;save blipmode setting

;*************************************************************
        ;Switch OFF System Variables

        (setvar "osmode" 0)
        ;Switch OFF snap

        (setvar "blipmode" 0)
        ;Switch OFF Blipmode

;*************************************************************
        ;Get User Inputs

        (initget (+ 1 2 4))
        ;check user input
        (setq lb (getdist "\nLength of Beam : "))
        ;get the length of the beam

        (initget (+ 1 2 4))
        ;check user input
        (setq hb (getdist "\nHeight of Beam : "))
        ;get the height of the beam

        (initget (+ 1 2 4))
        ;check user input
        (setq wt (getdist "\nFlange Thickness : "))
        ;get the thickness of the flange

        (initget (+ 1 2 4))
        ;check user input
        (setq ep (getdist "\nEnd Plate Thickness : "))
        ;get the thickness of the end plate

        (initget (+ 1 2 4))
        ;check user input
        (setq nl (getdist "\nLength of Notch : "))
        ;get the length of notch

        (initget (+ 1 2 4))
        ;check user input
        (setq nd (getdist "\nDepth of Notch : "))
        ;get the depth of the notch

        ;End of User Inputs
;*************************************************************
```

```
        ;Get Insertion Point

        (setvar "osmode" 32)
        ;switch ON snap

        (while
        ;start of while loop

        (setq ip (getpoint "\nInsertion Point : "))
        ;get the insertion point

        (setvar "osmode" 0)
        ;switch OFF snap

;*************************************************************
        ;Start of Polar Calculations

        (setq p2  (polar ip (dtr 180.0) (- (/ lb 2) nl)))
        (setq p3  (polar p2 (dtr 270.0) wt))
        (setq p4  (polar p2 (dtr 270.0) nd))
        (setq p5  (polar p4 (dtr 180.0) nl))
        (setq p6  (polar p5 (dtr 180.0) ep))
        (setq p7  (polar p6 (dtr 270.0) (- hb nd)))
        (setq p8  (polar p7 (dtr 0.0) ep))
        (setq p9  (polar p8 (dtr 90.0) wt))
        (setq p10 (polar p9 (dtr 0.0) lb))
        (setq p11 (polar p8 (dtr 0.0) lb))
        (setq p12 (polar p11 (dtr 0.0) ep))
        (setq p13 (polar p12 (dtr 90.0) (- hb nd)))
        (setq p14 (polar p13 (dtr 180.0) ep))
        (setq p15 (polar p14 (dtr 180.0) nl))
        (setq p16 (polar p15 (dtr 90.0) (- nd wt)))
        (setq p17 (polar p16 (dtr 90.0) wt))
        ;End of Polar Calculations
;*************************************************************
        ;Start of Command Function

        (command "Line" ip p2 p4 p6 p7 p12 p13 p15 p17 "c"
                 "Line" p3 p16 ""
                 "Line" p9 p10 ""
                 "Line" p5 p8 ""
                 "Line" p11 p14 ""
        )        ;End Command
        ;End of Command Function
;*************************************************************

        (setvar "osmode" 32)
        ;Switch ON snap

        );end of while loop
```

```
;*************************************************************
        ;Reset System Variable

        (setvar "osmode" oldsnap)
        ;Reset snap

        (setvar "blipmode" oldblipmode)
        ;Reset blipmode

;*************************************************************
        (princ)
        ;finish cleanly

)       ;end of defun

;*************************************************************
;This function converts Degrees to Radians.

(defun dtr (x)
        ;define degrees to radians function

        (* pi (/ x 180.0))
        ;divide the angle by 180 then
        ;multiply the result by the constant PI

)       ;end of function

;*************************************************************
(princ) ;load cleanly
;*************************************************************
```

The program will now repeat itself indefinitely, asking for an insertion point and drawing a beam, until the user presses enter.
This is how it works. The (while) function will continue to evaluate an expression until the expression evaluates to nil. As long as the user selects a point, the expression is true. But, when the user selects "Enter" the expression returns "nil" and the program moves out of the loop. (AutoLisp evaluates "Enter" as "nil")

I hope you have understood this tutorial and that it has given you a better understanding of AutoLisp. You will find a more detailed explanation of many of the functions used here in other tutorials within this manual

---

Loading AutoLISP Files.

Note!!

One of the most important things to remember about loading AutoLisp Routines is to ensure that your Lisp files and any support files (i.e DCL Files; DAT Files; etc) are in your AutoCad search path. (I dedicate a directory to all my Lisp files and relevant support files.

There are numerous ways of loading AutoLisp Files :

Command Line Loading.

The simplest is from the AutoCad command line.
The syntax for loading AutoLisp files is :

```
(load "filename")
```

The.lsp extension is not required.

Menu Loading.

The following code samples are one way of loading AutoLisp files from a menu.

Pull Down Menu's :

```
***POP12
T_Steel [Steel Menu]
T_Beams [Drawing Setup]^C^C^P+
(cond ((null C:DDSTEEL) (prompt "Please Wait...")(load "DDSTEEL"))) DDSTEEL
```

Toolbars :

```
***TOOLBARS
**STEEL
TB_DDSTEEL [_Button("Steel", "STEEL.bmp", "STEEL32.bmp")]^C^C^P+
(cond ((null C:ddsteel) (prompt "Please Wait...")(load "ddsteel"))) ddsteel
```

This method of loading Lisp files first checks to see if the routine is already loaded. If it is, it runs the routine. If it is not, it first loads the routine, then runs it. Clever Hey...

AcadDoc.Lsp File.

The AcadDoc.Lsp file is a useful way of loading a library of AutoLisp routines.
Each time you start a drawing AutoCad searches the library path for an AcadDoc.Lsp file. If it finds one, it loads the file into memory.

You could use the normal load function (load "filename") in your AcadDoc.Lsp file but if an error occurs whilst attempting to load one of your routines, the remainder of the file is ignored and is not loaded.
Therefore, you must use the *on failure* argument with the load function :

```
(load "Lispfile1" "\nLispfile1 not loaded")
(load "Lispfile2" "\nLispfile2 not loaded")
(load "Lispfile3" "\nLispfile3 not loaded")
```

The .MNL File

The other type of file that AutoCad loads automatically is the .MNL file.
If you have a partial menu file it can also have it's own .MNL file.
Just remember that the .MNL file must have exactly the same name as your partial menu file. (except for the .MNL extension, of course.)
You can load Lisp files from this file using the load function exactly the same as you did in the AcadDoc.Lsp file.

Command Autoloader

When you automatically load a command from your AcadDoc.Lsp file (or a .MNL file) the commands definition consumes your systems resources whether you actually use the command or not. The Autoload function makes a command available without loading the entire routine into memory.

```
(Autoload "Utils" '("Utils1" Utils2" "Utils3"))
(Autoload "DDSteel" '("DDSteel"))
```

This would automatically load the commands Utils1, Utils2 and Utils3 from the Utils.Lsp file and DDSteel from the DDSteel.Lsp file.

S::Startup Function.

If the user defined function S::Startup is included in the AcadDoc.lsp or a .MNL file, it is called when you enter a new drawing or open an existing drawing.

e.g. Say that you wanted to override the standard AutoCad LINE and COPY commands with versions of your own, your AcadDoc.Lsp file would something like this :

```
(defun C:LINE ()
   .....Your Definition.....
)
(defun C:COPY ()
   .....Your Definition.....
)
(defun S::Startup ()
    (command "Undefine" "LINE")
```

```
    (command "Undefine" "COPY")
)
```

Before the drawing is initialised, new definitions for LINE and COPY are defined. After the drawing is initialised, the S::Startup function is called and the standard definitions of LINE and COPY are undefined.

AutoCAD and Customizable Support Files

- Table of Customizable Support Files
- Order of support file loading when starting AutoCAD
- Automatically Load and Execute AutoLISP Routines
- Automatically Load ObjectARX Applications
- acad.lsp—Automatically Load AutoLISP
- acaddoc.lsp—Automatically Load AutoLISP
- Menu File Types
- Load Menu Files
- acad.mnl—Automatically Load AutoLISP Menu Functions
- S::STARTUP Function—Post-Initialization Execution
- Tips for Coding AutoLISP Startup Files
- Command Autoloader
- acadvba.arx—Automatically Load VBA
- acad.dvb—Automatically Load a VBA Project
- Automatically Loading a VBA Project
- Automatically Running a VBA Macro

Table of Customizable Support Files

AutoCAD uses support files for purposes such as storing menu definitions, loading AutoLISP and ObjectARX applications, and describing text fonts. Many support files are text files that you can modify with a text editor.

The following is a list of AutoCAD support files that can be edited. The support files are listed in alphabetical order by file extension. Please make backup copies of these files before modifying them.

Customizable support files	
File	**Description**
asi.ini	Database connectivity link conversion mapping file.
**.dcl*	AutoCAD Dialog Control Language (DCL) descriptions of dialog boxes.
**.lin*	AutoCAD linetype definition files.
acad.lin	The standard AutoCAD linetype library file.
acadiso.lin	The standard AutoCAD ISO linetype library file.
**.lsp*	AutoLISP program files.
acad.lsp	A user-defined AutoLISP routine that loads each time you start AutoCAD.
acaddoc.lsp	A user-defined AutoLISP routine that loads each time you start a drawing.
**.mln*	A multiline library file.
**.mnl*	AutoLISP routines used by AutoCAD menus. A MNL file must have the same file name as the MNU file it supports.
acad.mnl	AutoLISP routines used by the standard AutoCAD menu.
**.mns*	AutoCAD generated menu source files. Contains the command strings and macro syntax that define AutoCAD menus.

acad.mns	Source file for the standard AutoCAD menu.
*.mnu	AutoCAD menu source files. Contain the command strings and macro syntax that define AutoCAD menus.
acad.mnu	Source file for the standard AutoCAD menu.
*.pat	AutoCAD hatch pattern definition files.
acad.pat	The standard AutoCAD hatch pattern library file.
acadiso.pat	The standard AutoCAD ISO hatch pattern library file.
acad.pgp	The AutoCAD program parameters file. Contains definitions for external commands and command aliases.
acad.psf	AutoCAD PostScript Support file; the master support file for the PSOUT and PSFILL commands.
acad.rx	Lists ObjectARX applications that load when you start AutoCAD.
*.scr	AutoCAD script files. A script file contains a set of AutoCAD commands processed as a batch.
*.shp	AutoCAD shape/font definition files. Compiled shape/font files have the extension *.shx*.
acad.unt	AutoCAD unit definition file. Contains data that lets you convert from one set of units to another.

Order of support file loading when starting AutoCAD

You can understand the effects that one file may have on another if you know the order in which files are loaded when you start the software. For example, you have defined a function in an AutoLISP routine that is loaded from the *acad.lsp* file, but the function does not work when you start AutoCAD. This occurs because the function has been redefined by the *acaddoc.lsp* file, which is loaded after *acad.lsp*.

Following is a list of AutoCAD, Express Tools, and user-defined files in the order they are loaded when you first start the program.

File	For use by:
acad2000.lsp	AutoCAD
acad.rx	User
acad.lsp	User
acad2000doc.lsp	AutoCAD
acetutil.fas	Express Tools

acaddoc.lsp	User
mymenu.mnc	User
mymenu.mnl	User
acad.mnc	AutoCAD
acad.mnl	AutoCAD
acetmain.mnc	Express Tools
acetmain.mnl	Express Tools
s::startup	User

Note: If the user-defined function S::STARTUP is included in the *acad.lsp* or *acaddoc.lsp* file or a MNL file, the function is called when you enter a new drawing or open an existing drawing. Thus, you can include a definition of S::STARTUP in the LISP startup file to perform any setup operations.

Automatically Load and Execute AutoLISP Routines

As you build a library of useful AutoLISP routines, you may want to load them each time you run AutoCAD. You may also want to execute certain commands or functions at specific times during a drawing session.

AutoCAD loads the contents of four user-definable files automatically: acad.rx, *acad.lsp*, *acaddoc.lsp*, and the *.mnl* file that accompanies your current menu. By default, the *acad.lsp* loads only once, when AutoCAD starts, while *acaddoc.lsp* loads with each individual document (or drawing). This lets you associate the loading of the *acad.lsp* file with application startup, and the *acaddoc.lsp* file with document (or drawing) startup. The default method for loading these startup files can be modified by changing the setting of the *ACADLSPASDOC* system variable.

If one of these files defines a function of the special type *S::STARTUP*, this routine runs immediately after the drawing is fully initialized. The *S::STARTUP* function is described in S::STARTUP Function—Post-Initialization Execution. As an alternative, the *APPLOAD* command provides a Startup Suite option that loads the specified applications without the need to edit any files.

The *acad.lsp* and *acaddoc.lsp* startup files are not provided with AutoCAD. It is up to the user to create and maintain these files.

Automatically Load ObjectARX Applications

The *acad.rx* file contains a list of the ObjectARX program files that are loaded automatically when you start AutoCAD. You can edit this file with a text editor or word processor that produces files in ASCII text format. You can customize this file as you want, adding to or deleting from its contents and making the appropriate ObjectARX programs available for use. As an alternative, the *APPLOAD* command provides a Startup Suite option that loads the specified applications without the need to edit any files.

Because AutoCAD searches for the *acad.rx* file in the order specified by the library path, you can have a different *acad.rx* file in each drawing directory. This makes specific ObjectARX programs available for certain types of drawings. For example, you might keep 3D drawings in a directory called

***AcadJobs/3d_dwgs*. If that directory is set up as the current directory, you could copy the *acad.rx* file into that directory and modify it in the following manner:**

myapp1

otherapp

If you place this new *acad.rx* file in the *AcadJobs/3d_dwgs* directory and you start AutoCAD with that as the current directory, these new ObjectARX programs are then loaded and are available from the AutoCAD prompt line. Because the original *acad.rx* file is still in the directory with the AutoCAD program files, the default *acad.rx* file will be loaded if you start AutoCAD from another directory that does not contain an *acad.rx* file.

You can load ObjectARX programs from an *.mnl* file using the arxload function. This ensures that an ObjectARX program, required for proper operation of a menu, will be loaded when the menu file is loaded.

You can also autoload many ObjectARX-defined AutoCAD commands (see Command Autoloader and "*autoarxload*" in the *AutoLISP Reference*).

acad.lsp—Automatically Load AutoLISP

The *acad.lsp* file is useful if you want to load specific AutoLISP routines every time you start AutoCAD. When you start AutoCAD, the program searches the library path for an *acad.lsp* file. If it finds one, it loads the file into memory.

The *acad.lsp* file is loaded at each drawing session startup when AutoCAD is launched from the Windows desktop. Because the *acad.lsp* file is intended to be used for application-specific startup routines, all functions and variables defined in an *acad.lsp* file are only available in the first drawing. You will probably want to move routines that should be available in all documents from your *acad.lsp* file into the new *acaddoc.lsp* file.

The recommended functionality of *acad.lsp* and *acaddoc.lsp* can be overridden with the *ACADLSPASDOC* system variable. If the *ACADLSPASDOC* system variable is set to 0 (the default setting), the *acad.lsp* file is loaded just once; upon application startup. if *ACADLSPASDOC* is set to 1, the *acad.lsp* file is reloaded with each new drawing.

The *ACADLSPASDOC* system variable is ignored in *SDI* (single document interface) mode. When the *SDI* system variable is set to 1, the *LISPINIT* system variable controls reinitialization of AutoLISP between drawings. When *LISPINIT* is set to 1, AutoLISP functions and variables are valid in the current drawing only; each time you start a new drawing or open an existing one, all functions and variables are cleared from memory and the *acad.lsp* file is reloaded. Changing the value of *LISPINIT* when the SDI system variable is set to 0 has no effect.

The *acad.lsp* file can contain AutoLISP code for one or more routines, or just a series of load function calls. The latter method is preferable, because modification is easier. If you save the following code as an *acad.lsp* file, the files *mysessionapp1.lsp*, *databasesynch.lsp*, and *drawingmanager.lsp* are loaded every time you start AutoCAD.

(load "mysessionapp1")

(load "databasesynch")

(load "drawingmanager")

Note: Do not modify the reserved *acad2000.lsp* file. Autodesk provides the *acad2000.lsp* file, which contains AutoLISP defined functions that are required by AutoCAD. This file is loaded into memory immediately before the *acad.lsp* file is loaded.

acaddoc.lsp—Automatically Load AutoLISP

The *acaddoc.lsp* file is intended to be associated with each document (or drawing) initialization. This file is useful if you want to load a library of AutoLISP routines to be available every time you start a new drawing (or open an existing drawing). Each time a drawing opens, AutoCAD searches the library path for an *acaddoc.lsp* file. If it finds one, it loads the file into memory. The *acaddoc.lsp* file is always loaded with each drawing regardless of the settings of *ACADLSPASDOC* and *LISPINIT*.

Most users will have a single *acaddoc.lsp* file for all document-based AutoLISP routines. AutoCAD searches for an *acaddoc.lsp* file in the order defined by the library path; therefore, with this feature, you can have a different *acaddoc.lsp* file in each drawing directory, which would load specific AutoLISP routines for certain types of drawings or jobs.

The *acaddoc.lsp* file can contain AutoLISP code for one or more routines, or just a series of load function calls. The latter method is preferable, because modification is easier. If you save the following code as an *acaddoc.lsp* file, the files *mydocumentapp1.lsp*, *build.lsp*, and *counter.lsp* are loaded every time a new document is opened.

(load "mydocumentapp1")

(load "build")

(load "counter")

AutoCAD searches for an *acaddoc.lsp* file in the order defined by the library path; therefore, you can have a different *acaddoc.lsp* file in each drawing directory. You can then load specific AutoLISP routines for certain types of drawings or jobs.

Note: Do not modify the reserved *acad2000doc.lsp* file. Autodesk provides the *acad2000doc.lsp* file, which contains AutoLISP-defined functions that are required by AutoCAD. This file is loaded into memory immediately before the *acaddoc.lsp* file is loaded.

Menu File Types

The term *menu file* actually refers to the group of files that work together to define and control the appearance and functionality of the menu areas. The following table describes the AutoCAD menu file types.

AutoCAD menu files	
File type	**Description**
MNU	**Template menu file.**
MNC	**Compiled menu file. This binary file contains the command strings and menu syntax that defines the functionality and appearance of the menu.**
MNR	**Menu resource file. This binary file contains the bitmaps used by the menu.**
MNS	**Source menu file (generated by AutoCAD).**
MNT	**Menu resource file. This file is generated when the MNR file is unavailable, for example, read-only.**
MNL	**Menu LISP file. This file contains AutoLISP expressions that are used by the menu file and are loaded into memory when a menu file with the same file name is loaded.**

Load Menu Files

Use the *MENU* command to load a new menu. Use the *MENULOAD* and *MENUUNLOAD* commands to load and unload additional menus (called partial menus) and to add or remove individual menus from the menu bar.

AutoCAD stores the name of the last loaded menu in the system registry. This name is also saved with the drawing, but it is used only for backward compatibility. When you start AutoCAD, the last menu used is loaded. As of Release 14, AutoCAD no longer reloads the menu between drawings.

AutoCAD finds and loads the specified file according to the following sequence. (This sequence is also used when AutoCAD loads a new menu with the MENU command.)

- **AutoCAD looks for a menu source file (MNS) of the given name, following the library search procedure.**
- **If an MNS file is found, AutoCAD looks for a compiled menu file (.mnc) of the same name in the same directory. If AutoCAD finds a matching MNC file with the same or later date and time as the MNS file, it loads the MNC file. Otherwise, AutoCAD compiles the MNS file, generating a new MNC file in the same directory, and loads that file.**
- **If an MNS file is not found, AutoCAD looks for a compiled menu file (.mnc) of the given name, following the library search procedure. If AutoCAD finds the MNC file, it loads that file.**
- **If AutoCAD doesn't find either a MNS or a MNC file, it searches the library path for a menu template file (.mnc) of the given name. If this file is found, it compiles an MNC and MNS file, then loads the MNC file.**
- **If AutoCAD doesn't find any menu files of the given names, an error message is displayed and you are prompted for another menu file name.**
- **After finding, compiling, and loading the MNC file, AutoCAD looks for a menu LISP file (.mnl), using the library search procedure. If AutoCAD finds this file, it evaluates the AutoLISP expressions within that file.**

The acad.mnl file contains AutoLISP code used by the standard menu file, acad.mnu. The acad.mnl file is loaded each time the acad.mnu file is loaded.

Each time AutoCAD compiles an MNC file it generates a menu resource file (MNR) which contains the bitmaps used by the menu. The MNS file is an ASCII file that is initially the same as the MNU file (without comments or special formatting). The MNS file is modified by AutoCAD each time you make changes to the contents of the menu file through the interface (such as modifying the contents of a toolbar).

Although the initial positioning of the toolbars is defined in the MNU or MNS file, changes to the show/hide and docked/floating status or changes to the toolbar positions are recorded in the system registry. After an MNS file has been created, it is used as the source for generating future MNC, and MNR files. If you modify the MNU file after an MNS file has been generated, you must use the MENU command to explicitly load the MNU file so that AutoCAD will generate new MNS and MNC files and your changes will be recognized.

Note: If you use the interface to modify the toolbars, you should cut and paste the modified portions of the MNS file to the MNU file before deleting the MNS file.

The MENU command initially requests the MNS or MNC file. To reload a modified MNU file, choose the Menu Template item from the file type list, and then choose the MNU file from the list of files. Doing so protects the MNS file from accidentally being rebuilt, thus losing any toolbar or partial menu modifications done through the interface. While building and testing a menu file, you may find this procedure awkward. The following AutoLISP routine defines a new command, MNU, which reloads the current MNU file without going through all the prompts.

```
(defun C:MNU ()

  (command "_menu" (strcat (getvar "menuname") ".mnu"))

  (princ)

)
```

If you add this code to your acad.lsp file, the MNU command is automatically defined when you restart AutoCAD.

---

acad.mnl—Automatically Load AutoLISP Menu Functions

The other type of file that AutoCAD loads automatically accompanies your current menu file and has the extension *.mnl*. When AutoCAD loads a menu file, it searches for an MNL file with a matching file name. If it finds the file, it loads the file into memory.

This function ensures that AutoCAD loads the AutoLISP functions that are needed for proper operation of a menu. As an example, the standard AutoCAD menu, *acad.mnu*, relies on the file *acad.mnl* being loaded properly. This file defines numerous AutoLISP functions used by the menu. The MNL file is loaded after the *acaddoc.lsp* file.

Note: If a menu file is loaded with the AutoLISP command function (with syntax similar to (command "menu" "newmenu")), the associated MNL file is not loaded until the entire AutoLISP routine has run.

For example, if you create a custom menu called *newmenu.mnu* and you need to load three AutoLISP files (*new1.lsp*, *new2.lsp*, and *new3.lsp*) for the menu to work properly, you should create an ASCII text file named *newmenu.mnl* as follows:

(load "new1")

(load "new2")

(load "new3")

(princ "\nNewmenu utilities... Loaded.")

(princ)

In this example, calls to the princ function can be used to display status messages. The first use of princ displays the following on the command line:

Newmenu utilities... Loaded.

The second call to princ exits the AutoLISP function. Without this second call to princ, the message would be displayed twice. As mentioned previously, you can include the onfailure argument with calls to the load function as an extra precaution.

S::STARTUP Function—Post-Initialization Execution

The startup LISP files (*acad.lsp*, *acaddoc.lsp*, and *.mnl*) all load into memory before the drawing is completely initialized. Typically, this does not pose a problem, unless you want to use the command function, which is not guaranteed to work until after a drawing is initialized.

If the user-defined function *S::STARTUP* is included in an *acad.lsp*, *acaddoc.lsp* or a *.mnl* file, it is called when you enter a new drawing or open an existing drawing. Thus, you can include a definition of *S::STARTUP* in the LISP startup file to perform any setup operations.

For example, if you want to override the standard HATCH command by adding a message and then switching to the BHATCH command, use an *acaddoc.lsp* file that contains the following:

```
(defun C:HATCH ( )
  (alert "Using the BHATCH command!")
  (princ "\nEnter OLDHATCH to get to real HATCH command.\n")
  (command "BHATCH")
  (princ)
)
(defun C:OLDHATCH ( )
  (command ".HATCH")
```

```
  (princ)

)

(defun-q S::STARTUP ( )

  (command "undefine" "hatch")

  (princ "\nRedefined HATCH to BHATCH!\n")

)
```

Before the drawing is initialized, new definitions for HATCH and OLDHATCH are defined with the defun function. After the drawing is initialized, the *S::STARTUP* function is called and the standard definition of HATCH is undefined.

Note: To be appended, the *S::STARTUP* function must have been defined with the defun-q function rather than defun.

Because an *S::STARTUP* function can be defined in many places (an *acad.lsp*, *acaddoc.lsp*, *.mnl* file, or any other AutoLISP file loaded from any of these), it's possible to overwrite a previously defined *S::STARTUP* function. The following example shows one method of ensuring that your start-up function works with other functions.

```
(defun-q MYSTARTUP ( )

        ... your start-up function ...

)

(setq S::STARTUP (append S::STARTUP MYSTARTUP))
```

The previous code appends your start-up function to that of an existing *S::STARTUP* function, and then redefines the *S::STARTUP* function to include your start-up code. This works properly regardless of the prior existence of an *S::STARTUP* function.

Tips for Coding AutoLISP Startup Files

If an AutoLISP error occurs while you are loading a startup file, the remainder of the file is ignored and is not loaded. Files specified in a startup file that do not exist or that are not in the AutoCAD library path generally cause errors. Therefore, you may want to use the onfailure argument with the load function. The following example uses the onfailure argument:

```
(princ (load "mydocapp1" "\nMYDOCAPP1.LSP file not loaded."))

(princ (load "build" "\nBUILD.LSP file not loaded."))

(princ (load "counter" "\nCOUNTER.LSP file not loaded."))
```

(princ)

If a call to the load function is successful, it returns the value of the last expression in the file (usually the name of the last defined function or a message regarding the use of the function). If the call fails, it returns the value of the onfailure argument. In the preceding example, the value returned by the load function is passed to the princ function, causing that value to be displayed on the command line. For example, if an error occurs while AutoCAD loads the *mydocapp1.lsp* file, the princ function displays the following message and AutoCAD continues to load the two remaining files:

MYDOCAPP1.LSP file not loaded.

If you use the command function in an *acad.lsp*, *acaddoc.lsp* or MNL file, it should be called only from within a defun statement. Use the S::STARTUP function to define commands that need to be issued immediately when you begin a drawing session. The *S::STARTUP* function is described in *S::STARTUP Function—Post-Initialization Execution.*

Command Autoloader

When you automatically load a command using the load or command functions, the command's definition takes up memory whether or not you actually use the command. The AutoLISP autoload function makes a command available without loading the entire routine into memory. Adding the following code to your *acaddoc.lsp* file automatically loads the commands CMD1, CMD2, and CMD3 from the *cmds.lsp* file and the NEWCMD command from the *newcmd.lsp* file.

(autoload "CMDS" '("CMD1" "CMD2" "CMD3"))

(autoload "NEWCMD" '("NEWCMD"))

The first time you enter an automatically loaded command at the Command prompt, AutoLISP loads the entire command definition from the associated file. AutoLISP also provides the autoarxload function for ObjectARX applications. See "autoload" and "autoarxload" in the *AutoLISP Reference*.

acadvba.arx—Automatically Load VBA

You cannot load VBA until an AutoCAD VBA command is issued. If you want to load VBA automatically every time you start AutoCAD include the following line in the *acad.rx* file:

acadvba.arx

You can automatically run a macro in the *acad.dvb* file by naming the macro AcadStartup. Any macro in your *acad.dvb* file called AcadStartup automatically executes when VBA loads.

acad.dvb—Automatically Load a VBA Project

The *acad.dvb* file is useful if you want to load a specific VBA project that contains macros you want each

time you start AutoCAD. Each time you start a new AutoCAD drawing session, AutoCAD searches for the *acad.dvb* file and loads it.

If you want a macro in your *acad.dvb* file to run each time you start a new drawing or open an existing one, add the following code to your *acaddoc.lsp* file:

```
(defun S::STARTUP()

  (command "_-vbarun" "updatetitleblock")

)
```

Automatically Loading a VBA Project

There are two different ways to load a VBA project automatically:

- **When VBA is loaded it will look in the AutoCAD directory for a project named *acad.dvb*. This file will automatically load as the default project**
- **Any project other than the default, *acad.dvb*, can be used by explicitly loading that project at startup using the VBALOAD command. The following code sample uses the AutoLISP startup file to load VBA and a VBA project named *myproj.dvb* when AutoCAD is started. Start *notepad.exe* and create (or append to) *acad.lsp* the following lines:**

```
(defun S::STARTUP()

(command "_VBALOAD" "myproj.dvb")

)
```

Automatically Running a VBA Macro

You can automatically run any macro in the *acad.dvb* file by calling it with the command line version of VBARUN from an AutoCAD startup facility like *acad.lsp*. For example, to automatically run the macro named drawline, first save the drawline macro in the *acad.dvb* file. Next, invoke *notepad.exe* and create (or append to) *acad.lsp* the following lines:

```
(defun S::STARTUP()

    (command "_-vbarun" "drawline")

)
```

You can cause a macro to run automatically when VBA loads by naming the macro AcadStartup. Any macro in your *acad.dvb* file called AcadStartup will automatically get executed when VBA loads.

Environment Variables Listing.

Compiled and kindly donated by Stig Madsen.

Remember that environment variables are dependent on .. well, the environment, so each may or may not apply to a certain setup.
Some are almost described, some are definitely not. The first are OS dependent, the rest are AutoCAD dependent.

System related

(getenv "Path") ;string System search paths

(getenv "COMSPEC") ;string Cmd.exe path

(getenv "UserName");string User logon name

(getenv "Temp") ;string Temp path

(getenv "TMP") ;string Temp path

(getenv "ComputerName");string Computer name

(getenv "Windir") ;string Windows path

(getenv "OS") ;string Operating system

(getenv "UserProfile");string Current user profile path

(getenv "Pathext") ;string Exec extensions

(getenv "SystemDrive");string System drive

(getenv "SystemRoot");string System root path

(getenv "MaxArray");integer

General

(getenv "ACAD") ;string Support search paths

(getenv "ANSIHatch");string Pattern file for ANSI setup 1)

(getenv "ANSILinetype");string Linetype file for ANSI setup 1)

(getenv "ISOHatch");string Pattern file for ISO setup 1)

(getenv "ISOLinetype");string Linetype file for ISO setup 1)

(getenv "StartUpType");string Current default for StartUp dialog

(getenv "acet-MenuLoad");string Loading of Express Tools menu

(getenv "Measureinit");string MEASUREINIT

(getenv "InsertUnitsDefSource");integer INSUNITSDEFSOURCE

(getenv "InsertUnitsDefTarget");integer INSUNITSDEFTARGET

(getenv "acet-Enable");string

(getenv "LastTemplate");string Last DWT used

(getenv "AcetRText:type");string Current default for RTEXT "Diesel"

(getenv "Pickstyle");integer

(getenv "Coords") ;integer

(getenv "ShowProxyDialog");integer

(getenv "Osmode") ;integer

(getenv "EdgeMode");integer

(getenv "PAPERUPDATE");integer

(getenv "ACADPLCMD");string Plotter command string

(getenv "ImageHighlight");integer

(getenv "Attdia") ;integer

(getenv "Attreq") ;integer

(getenv "Delobj") ;integer

(getenv "Dragmode");integer

(getenv "UseMRUConfig");integer

(getenv "PLSPOOLALERT");integer

(getenv "PLOTLEGACY");integer

(getenv "PSTYLEPOLICY");integer

(getenv "OLEQUALITY");integer

(getenv "Anyport") ;integer

(getenv "Validation Policy");integer

(getenv "Validation Strategy");integer

(getenv "CommandDialogs");integer CMDDIA

(getenv "TempDirectory");string Temp dir

(getenv "PlotSpoolerDirectory");string Spooler dir

(getenv "DefaultLoginName");string Default login

(getenv "MenuFile");string Default menu path

(getenv "NetLocation");string Default URL

(getenv "ACADDRV") ;string Driver path

(getenv "ACADHELP");string Help path

(getenv "PrinterConfigDir");string Plotter path

(getenv "PrinterStyleSheetDir");string Plot styles path

(getenv "PrinterDescDir");string Plotter driver path

(getenv "NewStyleSheet");string Default .stb/.ctb file

(getenv "DefaultFormatForSave");integer Default saveas

(getenv "DefaultConfig");string Default pc3

(getenv "LastModifiedConfig");string Last pc3

(getenv "MRUConfig");string pc3?

(getenv "ACADLOGFILE");string Logfile

(getenv "MaxDwg") ;integer

(getenv "AVEMAPS") ;string Texture files path

(getenv "TemplatePath");string Templates path

(getenv "DatabaseWorkSpacePath");string Data Links path

(getenv "DefaultPlotStyle");string e.g. "ByLayer"

(getenv "DefaultLayerZeroPlotStyle");string e.g."Normal"

(getenv "LineWeightUnits");integer

(getenv "LWDEFAULT");integer Default lineweight

(getenv "CustomColors");integer

(getenv "Blipmode");integer

(getenv "ToolTips");string

1) used by MEASUREINIT and MEASUREMENT sysvars

Editor Configuration

(getenv "SDF_AttributeExtractTemplateFile");string ??

(getenv "AutoSnapPolarAng");string POLARANG

(getenv "AutoSnapPolarDistance");string POLARDIST

(getenv "AutoSnapPolarAddAng");string POLARADDANG

(getenv "AutoSnapControl");integer AUTOSNAP

(getenv "AutoSnapTrackPath");integer TRACKPATH

(getenv "PickBox") ;integer PICKBOX

(getenv "AutoSnapSize");integer

(getenv "PickFirst");integer PICKFIRST

(getenv "PickAuto");integer PICKAUTO

(getenv "MenuOptionFlags");integer MENUCTL

(getenv "FontMappingFile");string

(getenv "LogFilePath");string

(getenv "PSOUT_PrologFileName");string

(getenv "MainDictionary");string

(getenv "CustomDictionary");string

(getenv "MTextEditor");string

(getenv "XrefLoadPath");string

(getenv "SaveFilePath");string

(getenv "AcadLspAsDoc");string

Drawing Window

(getenv "Background");integer Background color

(getenv "Layout background");integer PS Background color

(getenv "XhairPickboxEtc");integer Crosshair color

(getenv "LayoutXhairPickboxEtc");integer PS Crosshair color

(getenv "Autotracking vector");integer Autotracking vector color

(getenv "MonoVectors");integer

(getenv "FontFace");string Screen Menu

(getenv "FontHeight");integer

(getenv "FontWeight");integer

(getenv "FontItalic");integer

(getenv "FontPitchAndFamily");integer

(getenv "CursorSize");integer

(getenv "HideWarningDialogs");integer:00000008 <- hit

(getenv "SDIMode") ;integer:00000000 <- hit

Command Line Windows

(getenv "CmdLine.ForeColor");integer

(getenv "CmdLine.BackColor");integer

(getenv "TextWindow.ForeColor");integert

(getenv "TextWindow.BackColor");integer

(getenv "CmdLine.FontFace");string

(getenv "CmdLine.FontHeight");integer

(getenv "CmdLine.FontWeight");integer

(getenv "CmdLine.FontItalic");integer

(getenv "CmdLine.FontPitchAndFamily");integer

(getenv "TextWindow.FontFace");string

(getenv "TextWindow.FontHeight");integer

(getenv "TextWindow.FontWeight");integer

(getenv "TextWindow.FontItalic");integer

(getenv "TextWindow.FontPitchAndFamily");integer

Migrating 2000's Express Tools To 2002

If you are upgrading from AutoCAD 2000 to AutoCAD 2002 and also have Express Tools installed, you should choose to install without uninstalling the older software version first. This option will ensure that Express Tools migrate into AutoCAD 2002 without difficulty.
Users who choose to completely uninstall AutoCAD 2000 prior to installing AutoCAD 2002, will need to first backup the Express Tools library as the library is not included with AutoCAD 2002.

Step by Step Guide

1 Prior to uninstalling AutoCAD 2000, please make a backup copy of the following Express Tools-related files found in the following directories:

- **Express*.***
- **Support\acettest.fas**
- **Help\acetmain.hlp**
- **Help\acetmain.cnt**
- **Help\acetfaq.hlp**

2 Install AutoCAD 2002.

3 Restart your PC.

3 Create a subdirectory called Express under the directory you installed AutoCAD 2002. For example if you installed AutoCAD on the *C:*\ drive in the directory *C:\Program Files\AutoCAD 2002*, create the directory *C:\Program Files\AutoCAD 2002\Express*.

4 Copy all of the Express Tools files from *Express.* backed up in Step 1 into this new directory you created in Step 3.**

5 Copy *acettest.fas* to the AutoCAD 2002 Support Files subdirectory. Following the example above, if you have installed AutoCAD on the *C:*\ drive in the *C:\Program Files\AutoCAD 2002* directory, the Support files directory would be *C:\Program Files\AutoCAD 2002\Support.*

6 Copy *acetmain.hlp, acetmain.cnt, acetfaq.hlp* to the AutoCAD 2002 Help subdirectory. Again, following the example directories above, the help directory would be *C:\Program Files\AutoCAD 2002\Help.*

7 At the command line, enter the *EXPRESSMENU* command by typing *EXPRESSMENU* at the command line.

The Express Tools menu will now load automatically in AutoCAD 2002 and

Creating Menu's

You can write or, obtain, as many AutoLISP routines as you like, but it's still a pain in the bum to have to type in something like :

(load "DDStruc_Steel_Ver2")

To have to type this in, or even remember the name every time you need to load or, run the routine, is just basically daft and un-productive.
You could, load all your routines from the AcadDoc.Lsp file. This though, would mean that they are all in memory, chewing up your system resources.
What you need to be able to do is to load/run the routines from the AutoCAD menu. Not so long ago, in earlier releases, the only option you had was to modify the standard AutoCAD menu. Not anymore. Now you can create a custom menu known as a "Partial Menu" and install this to run side by side with the standard AutoCAD menu.

This Tutorial will take you through all the steps of developing a fully, functional Custom Standard Menu.

Note : I will only be covering Pull Down Menu's, Image Menu's and Toolbars in this tutorial. If you need information on Screen, Button or Tablet Menu's, then please refer to the AutoCAD Customization Manual.

To get started let's begin with designing a simple Pull Down or Pop Menu.
Fire up your text editor and type in the following saving the file as Test.Mnu :

(Please ensure that you save all files to a directory in the AutoCAD Search Path.)

```
***MENUGROUP=TEST                    //menu name

***POP1                              //pull down name
P1-1[Test Menu]                      //pull down label
P1-2[Line]                           //menu items preceeded with ID
P1-3[Copy]
P1-4[Move]
P1-5[Zoom]
```

The first line :

*****MENUGROUP=TEST**

is the name of the Partial Menu.
The second line :

*****POP1**

is the name of the Drop Down or POP menu.
The third line consists of 2 parts :

P1-1

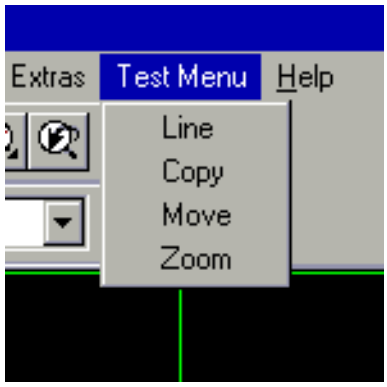
This is the "Name Tag" or "ID" of the menu item and allows you to access the menu item programmatically. The second part :

[Test-Menu]

is the menu label that appears in the Pull Down Menu.

The first label in a pull down menu always defines the menu bar title whilst, the succeeding labels define menu and sub-menu items.

Now we need to load the menu.
A new menu item will appear on the menu bar entitled "Test Menu".
It should look like this :

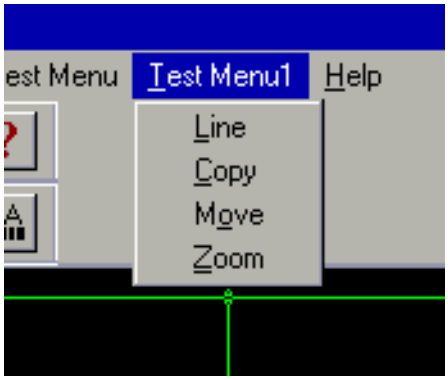


O.K. Now we've got our menu to display but, there's a problem. It doesn't do anything!! Let's make it functional.

Create a new menu file entitled TEST1.MNU and type in the following:

```
***MENUGROUP=TEST1

***POP1
P1-1[&Test Menu1]
P1-2[&Line]^C^CLine
P1-3[&Copy]^C^CCopy
P1-4[M&ove]^C^CMove
P1-5[&Zoom]^C^CZoom
```

Load this menu, following the same routine as you did for the first menu.
"Test Menu1" will appear in the menu bar and your new pull down menu should look like this :



Let's have a close look at one of the menu items :

P1-2[&Line]^C^CLine

The first part, P1-1 is, of course, the menu item ID.

The second part, [&Line] is the menu label. But did you notice something different? What is the '&' character doing in front of the 'L'?
If you precede any letter in the menu item label with '&', this will define the letter as the shortcut key to this menu item.
(There are other special label characters but, we will discuss these at a later stage.) In other words, when you press 'ALT L' the Line Menu Item will be chosen and any action affiliated to it will be triggered.

Following the item label, each menu item can consist of a command, parameter, or a sequence of commands and parameters. Let's look at the Line menu item macro.

^C^CLine

Just in case we have a previous incomplete command, we use the string ^C^C to start our menu macro. This is exactly the same as pressing CTRL+C or ESC, twice on the keyboard. We use ^C twice because some AutoCAD commands need to be cancelled twice before they return to the Command prompt.
(e.g. The Dim Command.)
We immediately follow this sequence with our AutoCAD command, Line.
When a menu item is selected, AutoCAD places a blank after it. A blank in a menu macro is interpreted as ENTER or SPACEBAR. In effect, this is exactly the same as typing, at the command prompt "Line" followed by ENTER.

More about Menu Macro's on the next.....

Menu Macro's

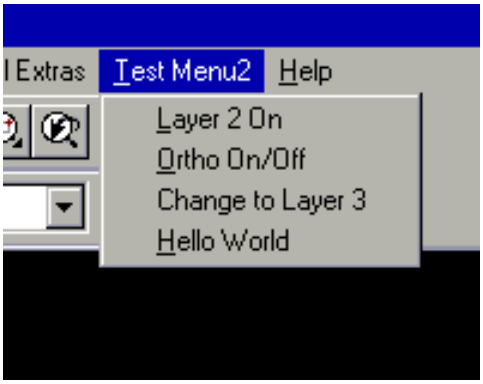
Let's look at some Special Menu Characters :

[ ]	**Encloses a Menu Label.**
;	**Enter.**
Space	**Enter or Spacebar.**
****	**Pauses for user input.**
_	**Translates AutoCAD commands.**
+	**Continues menu macro to next line.**
***^C^C**	**Prefix for repeating Item.**
$	**Character Code that loads a menu section.**
^B	**Toggels Snap/Off.**
^C	**Cancels command.**
^D	**Toggles Coords.**
^E	**Sets the next Isometric Plane.**
^G	**Toggles Grid On/Off.**
^H	**Issues Backspace.**
^O	**Toggles Ortho On/Off.**
^P	**Toggles Menuecho On/Off.**
^Q	**Echoes all prompts.**
^T	**Toggles Tablet On/Off.**
^V	**Changes current Viewport.**
^Z	**Null Character.**

I don't really want to get too involved with Menu Macro's, but here is a few sample Macro's to show you the general idea :

```
***MENUGROUP=TEST2

***POP1
P1-1[&Test Menu2]
P1-2[&Layer 2 On]^C^CLayer;M;2;;
P1-3[&Ortho On/Off]^C^C^O
P1-4[Change to Layer 3]*^C^CChange;\;P;LA;3;;
P1-5[&Hello World](alert "Hello World")
```

Your Menu should look like this :



The first Menu Item is again, the menu label.
The second item changes the current layer to layer 2.

(Layer Enter Make Enter 2 Enter Enter).
The third item simply toggles Ortho On or Off.
The fourth item let's you select an object and changes it to layer 3.
(Change Enter Pause Enter Properties Enter Layer Enter 3 Enter Enter).
The * prefix will force the macro to repeat until the user hits CTRL-C or ESC.
The fourth item demonstrates the use of AutoLISP within a menu item.

Menu macro's can get quite long and complicated but my advice to you is, rather write an AutoLISP routine than try and design complicated macro's.
Look at this for a macro :

```
[Box](setq a (getpoint "Enter First Corner: "));\ +
(setq b (getpoint "Enter Second Corner: "));\+
pline !a (list (car a)(cadr b))!b (list (car b) (cadr a))c;
```

Crazy, Hey. As I said, rather write an AutoLISP routine. You can do a hell of a lot more and, create a much more professional routine using AutoLISP than you ever will trying to write super-duper, 20 line macro's.

Just one small point about the above macro. Did you notice the use of the special character + at the end of some of the lines. This allows the macro to continue to the next line.

Anyway, enough about menu macro's. Let's have a look at some special label characters that you can use in pull-down menu's.

`--`	Item label that expands to become a separator line.
`+`	Continues macro to next line.
`->`	Label prefix that indicates that the pull-down menu item has a submenu
`<-`	Label prefix that indicates that the pull-down menu item is the last item in a submenu.
`<-<-..`	Label prefix that indicates that the pull-down menu item is the last item in the submenu, and terminates the parent menu. One <- is required to each terminate each parent menu.
`~`	Lable prefix that disables a menu item.
`!.`	Label prefix that marks a menu item.
`&`	Label prefix that defines shortcut key.

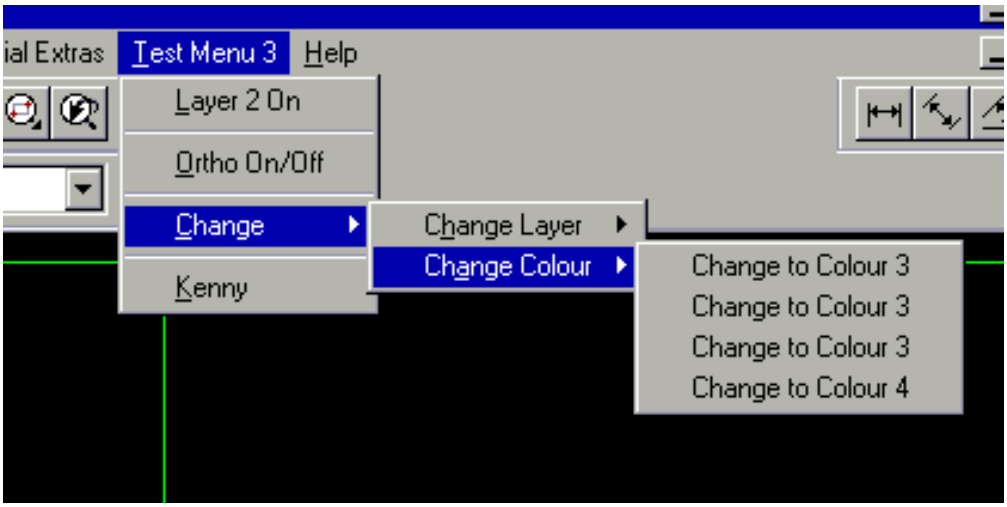
Next we will have a look at an example menu using some of these special characters.

Before loading the following menu, you should really un-load any of the test menu's that you still have loaded. If you don't, you just might run out of space on your menu bar.

OK, I see you're back. Now load the following menu file :

```
***MENUGROUP=TEST3

***POP1
P1-1[&Test Menu 3]
P1-2[&Layer 2 On]^C^CLayer;M;2;;                          //menu item
P1-3[--]                                                  //divider
P1-4[&Ortho On/Off]^C^C^O                                 //menu item
P1-5[--]                                                  //divider
P1-6[->&Change]                                           //submenu label
P1-7[->C&hange Layer]                                     //submenu label
P1-8[Change to Layer 1]^C^CChange;\;P;LA;1;;              //submenu items
P1-9[Change to Layer 2]^C^CChange;\;P;LA;2;;
P1-10[Change to Layer 3]^C^CChange;\;P;LA;3;;
P1-11[<-Change to Layer 4]^C^CChange;\;P;LA;4;;           //submenu terminator
P1-12[->Ch&ange Colour]                                   //submenu label
P1-13[Change to Colour 3]^C^CChange;\;P;C;1;;             //submenu items
P1-14[Change to Colour 3]^C^CChange;\;P;C;2;;
P1-15[Change to Colour 3]^C^CChange;\;P;C;3;;
P1-16[<-<-Change to Colour 4]^C^CChange;\;P;C;4;;         //submenu terminator
P1-17[--]                                                 //divider
P1-18[&Kenny](alert "Kenny is Handsome")                  //menu item
```

Your menu, hopefully, should look like this :



Right, enough of pull-downs for the meantime. Let's now have a look at Image Menu's.

Firstly, we need to create a couple of slides to display in our image menu.
Let's create some slides named D1 to D8, DDOOR and HDDOOR.
You can create these slides yourself using any object that you wish, as long as they are named

as stated. (I need the server space and drawing files are rather big.) Now we need to create a file library as a container for these slides.

Locate the Slidelib.exe function. It is normally in your Acad Support directory. Copy it to the same directory as your slides.
Now create a text file named Slidelist.LST and in it, make a list of all the slide names. (Remember to check spelling and case.)

```
D1
D2
D3
D4
D5
D6
D7
D8
DDOOR
HDDOOR
```

Make sure that this file is also in the same directory as the slides.
Now go to DOS. (Remember that 'black' place?)
At the DOS prompt find your way to the directory where your slides, Slidelib.exe and Slidelist.LST are located.
Type :

```
slidelib DOORS < Slidelist.LST then ENTER.
```

If you have done everything correct, DOORS.SLB should be created.
Next, we need to add an Image section to our menu file :

```
***MENUGROUP=TEST4

***POP1
P1_1[&Test Menu 4]
P1_2[&Layer 2 On]^C^CLayer;M;2;;
P1_3[--]
P1_4[&Ortho On/Off]^C^C^O
P1_5[--]
P1_6[->&Change]
P1_7[->C&hange Layer]
P1_8[Change to Layer 1]^C^CChange;\;P;LA;1;;
P1_9[Change to Layer 2]^C^CChange;\;P;LA;2;;
P1_10[Change to Layer 3]^C^CChange;\;P;LA;3;;
P1_11[<-Change to Layer 4]^C^CChange;\;P;LA;4;;
P1_12[->Ch&ange Colour]
P1_13[Change to Colour 3]^C^CChange;\;P;C;1;;
P1_14[Change to Colour 3]^C^CChange;\;P;C;2;;
P1_15[Change to Colour 3]^C^CChange;\;P;C;3;;
P1_16[<-<-Change to Colour 4]^C^CChange;\;P;C;4;;
P1_17[--]
P1_18[&Kenny](alert "Kenny is Handsome")
P1_19[--]
P1_20[Image Menu]^C^C$I=TEST4.DOORS $I=*                    //calls Image Menu
```

```
***IMAGE
**DOORS
[DOORS Created by Kenny Ramage ]
[DOORS(D1,DOOR1)]INSERT;*D1;\;;
[DOORS(D2,DOOR2)]INSERT;*D2;\;;
[DOORS(D3,DOOR3)]INSERT;*D3;\;;
[DOORS(D4,DOOR4)]INSERT;*D4;\;;
[DOORS(D5,DOOR5)]INSERT;*D5;\;;
[DOORS(D6,DOOR6)]INSERT;*D6;\;;
[DOORS(D7,DOOR7)]INSERT;*D7;\;;
[DOORS(D8,DOOR8)]INSERT;*D8;\;;
[DOORS(DDOOR,DOUBLE DOOR)]INSERT;*DDOOR;\;;
[DOORS(HDDOOR,DOOR & HALF)]INSERT;*HDDOOR;\;;
[ FITTINGS]$I=KENNY.FITTINGS $I=*
```

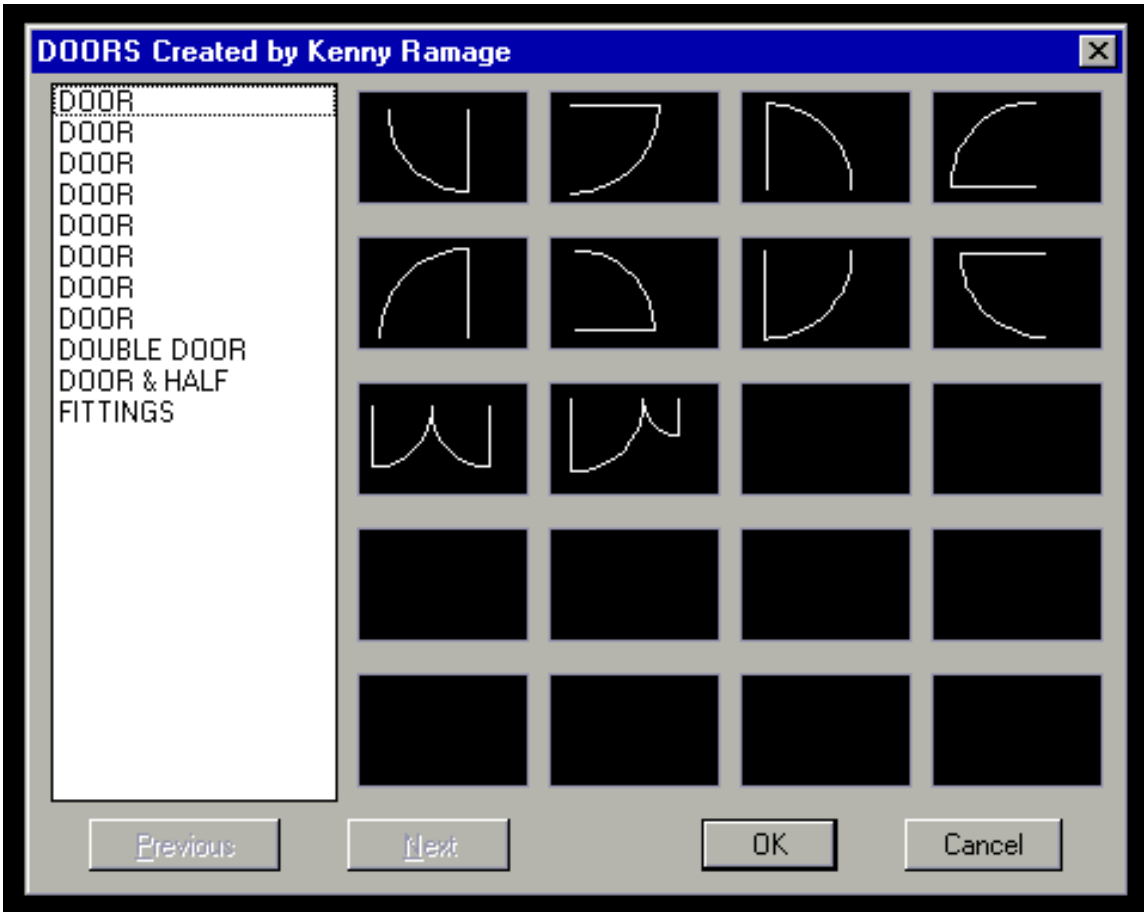
The first line, ***IMAGE, defines the Image section of the menu.
The second line, **DOORS, is the name of the Image section submenu.
The third line is the title of the label that appears at the top.
The following lines get a bit more complicated so, let's dissect them.

[DOORS(D1,DOOR)]INSERT;*D1;\;;

[DOORS is the name of the slide library.
(D1, is the name of the specific slide.
,DOOR1)] is the label that is displayed in the list box.
INSERT;*D1;\;; is the menu macro.

Your Pull Down menu should now look like this :

And your Image menu like this (with different images of course) :



Did you notice the method of calling the image menu?

P1_20[Image Menu]^C^C$I=TEST4.DOORS $I=*

Good, at least some one is awake ;-)
Phew...(Wipe's sweat off brow)....Time for a break.
Next we'll move on to Custom Toolbars.

Custom Toolbars

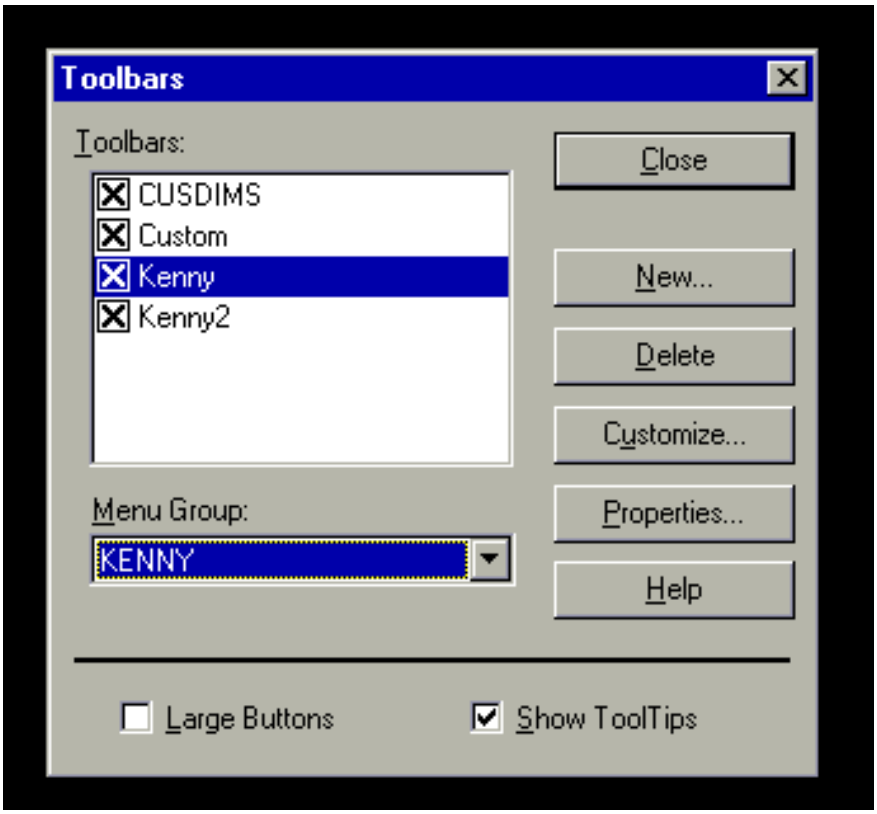
The easiest way of creating a custom toolbar is to use the AutoCAD interface.
First, copy and rename Test4.MNU to Test5.MNU. Edit the menu file so that it looks like this :

```
***MENUGROUP=TEST5

***POP1
P1_1[&Test Menu 5]
P1_2[&Layer 2 On]^C^CLayer;M;2;;
P1_3[--]
P1_4[&Ortho On/Off]^C^C^O
P1_5[--]
P1_6[->&Change]
P1_7[->C&hange Layer]
P1_8[Change to Layer 1]^C^CChange;\;P;LA;1;;
P1_9[Change to Layer 2]^C^CChange;\;P;LA;2;;
P1_10[Change to Layer 3]^C^CChange;\;P;LA;3;;
P1_11[<-Change to Layer 4]^C^CChange;\;P;LA;4;;
P1_12[->Ch&ange Colour]
P1_13[Change to Colour 3]^C^CChange;\;P;C;1;;
P1_14[Change to Colour 3]^C^CChange;\;P;C;2;;
P1_15[Change to Colour 3]^C^CChange;\;P;C;3;;
P1_16[<-<-Change to Colour 4]^C^CChange;\;P;C;4;;
P1_17[--]
P1_18[&Kenny](alert "Kenny is Handsome")
P1_19[--]
P1_20[Image Menu]^C^C$I=TEST4.DOORS $I=*

***IMAGE
**DOORS
[DOORS Created by Kenny Ramage ]
[DOORS(D1,DOOR1)]INSERT;*D1;\;;
[DOORS(D2,DOOR2)]INSERT;*D2;\;;
[DOORS(D3,DOOR3)]INSERT;*D3;\;;
[DOORS(D4,DOOR4)]INSERT;*D4;\;;
[DOORS(D5,DOOR5)]INSERT;*D5;\;;
[DOORS(D6,DOOR6)]INSERT;*D6;\;;
[DOORS(D7,DOOR7)]INSERT;*D7;\;;
[DOORS(D8,DOOR8)]INSERT;*D8;\;;
[DOORS(DDOOR,DOUBLE DOOR)]INSERT;*DDOOR;\;;
[DOORS(HDDOOR,DOOR & HALF)]INSERT;*HDDOOR;\;;
[ FITTINGS]$I=KENNY.FITTINGS $I=*
```

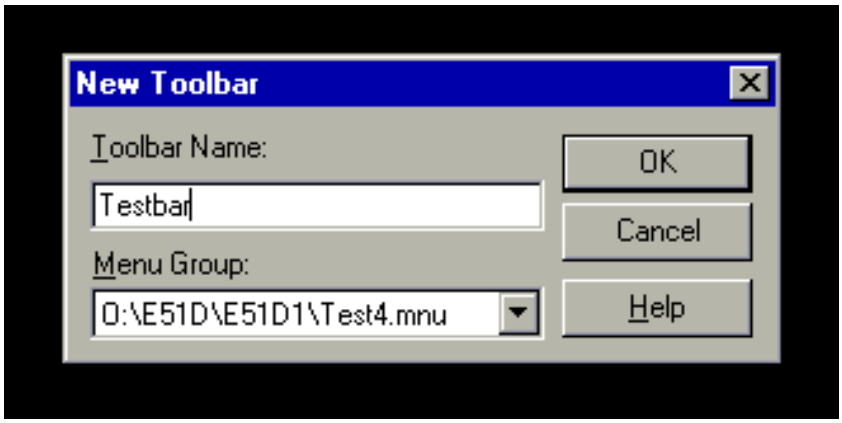
Open AutoCAD and load this new menu file.

Now, to create a new toolbar, follow these steps :

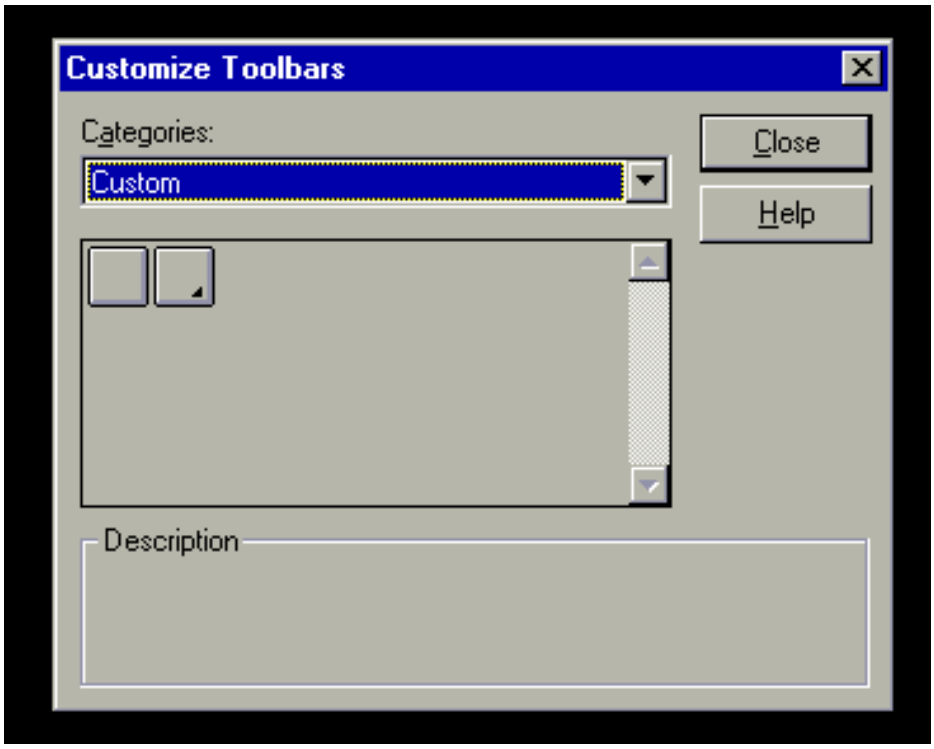
Right Click on any toolbar.
The Toolbars Dialogue will open.



Select "New".
The New Toolbar dialogue will open



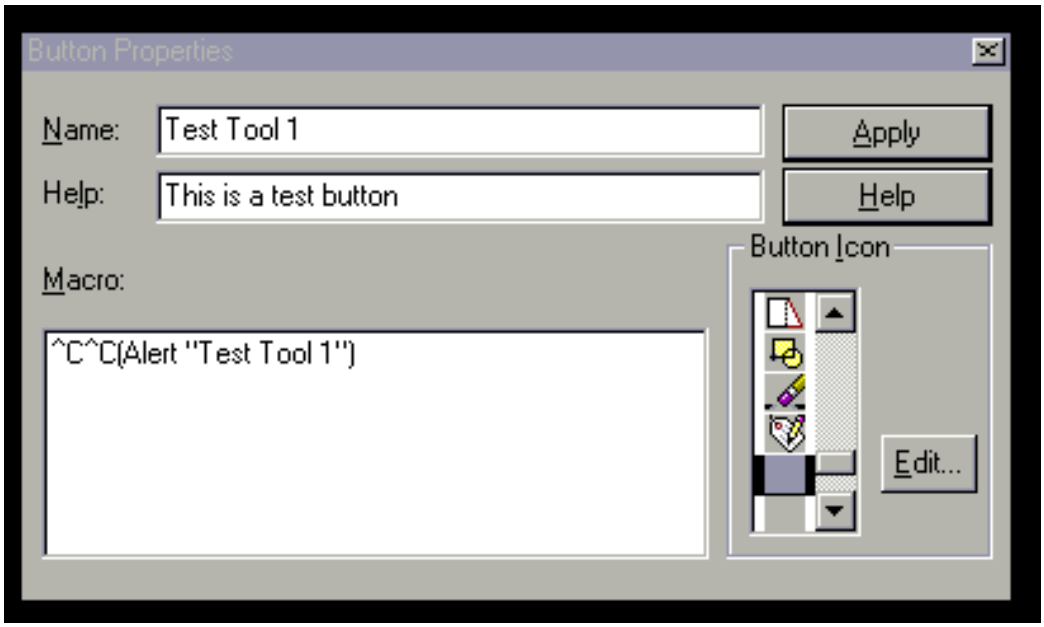
In the "Toolbar Name" edit box enter "Testbar" and from the "Menu Group" drop down list select Test5.mnu
Select O.K.
A small empty toolbar will appear on your screen.
Select "Customize" from the "Toolbars" dialogue.
The "Customize Toolbars" dialogue will open.



In the "Catagories" list box select "Custom".
Select the blank tile then drag and drop it onto your empty toolbar.
Repeat this so that you have 3 blank tiles on your toolbar.
Your toolbar should look like this :

Select "Close"
You now have a toolbar with 3 buttons that do exactly nothing!!

Right Click on the first blank tile.
The "Button Properties" dialogue will open.

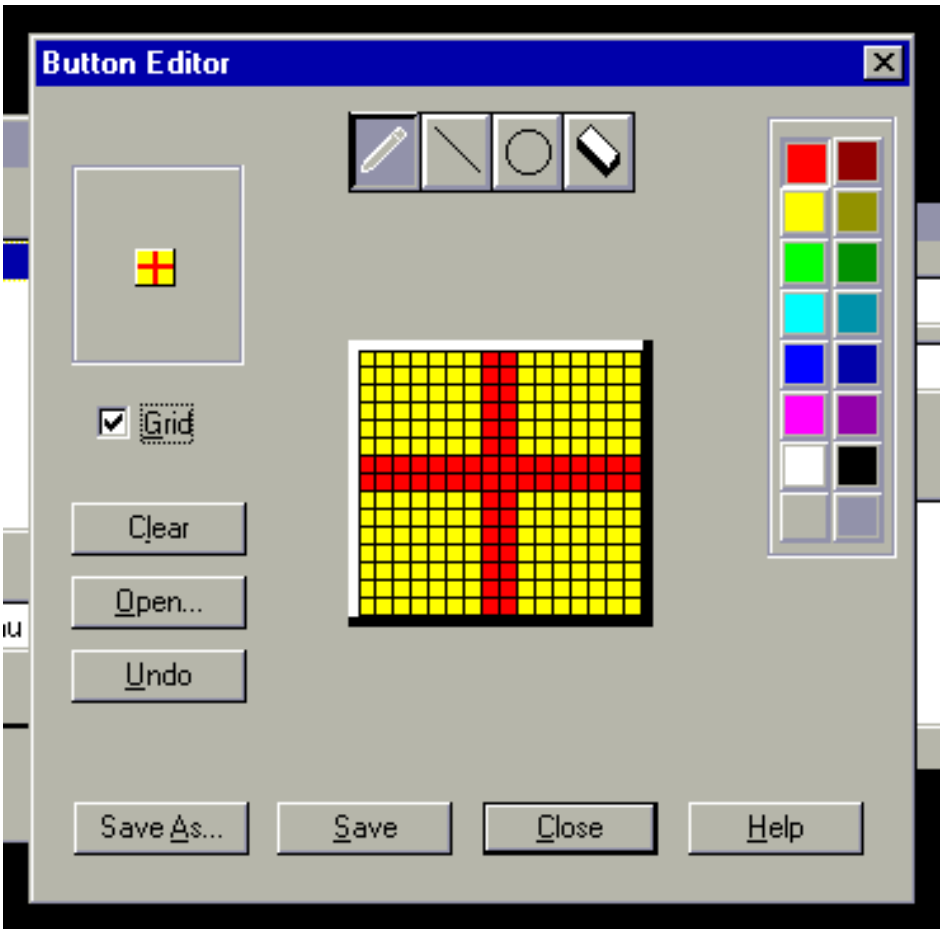


In the "Name" edit box type "Test Tool 1".
This will appear as a tooltip.
In the "Help" edit box type "This is Test Button 1".
This will appear in the status bar.
In the "Macro Edit" edit box enter :

^C^C(Alert "Test Tool 1")

This is the action assigned to this button.

Now, from the "Button Icon" list, select the first blank tile, then "Edit".
The "Button Editor" will open.



Select "Grid" to place a grid over your button.
Using the edit tools and the colour palette, draw the image that you would like to appear on your button.
When you are happy select "Save" then "Close".
In the "Buttons Properties" dialogue select "Apply".
You will notice your completed button appear in your toolbar.

Right Click on the second button and repeat the process, naming the button "Test Tool 2".
Then again for the third button naming it "Test Tool 3".

When you have finished and have closed all dialogue boxes, your toolbar should look something like this :

Select any of the buttons on the toolbar and you should get an alert message.
Now Exit AutoCAD.

Open Test5.mnu
Do you notice that nothing has changed and there is no Toolbar section in the MNU file?

The reason for this is because when you Add, Move or Edit any toolbars using the AutoCAD interface, the results are written to the MNS file and not to the MNU file. To update the MNU file and make our changes permanent, we need to copy and paste the toolbars section from the MNS file into the MNU file.
Right, let's do that. Your MNU file should now look like this :

```
***MENUGROUP=TEST5

***POP1
P1_1[&Test Menu 5]
P1_2[&Layer 2 On]^C^CLayer;M;2;;
P1_3[--]
P1_4[&Ortho On/Off]^C^C^O
P1_5[--]
P1_6[->&Change]
P1_7[->C&hange Layer]
P1_8[Change to Layer 1]^C^CChange;\;P;LA;1;;
P1_9[Change to Layer 2]^C^CChange;\;P;LA;2;;
P1_10[Change to Layer 3]^C^CChange;\;P;LA;3;;
P1_11[<-Change to Layer 4]^C^CChange;\;P;LA;4;;
P1_12[->Ch&ange Colour]
P1_13[Change to Colour 3]^C^CChange;\;P;C;1;;
P1_14[Change to Colour 3]^C^CChange;\;P;C;2;;
P1_15[Change to Colour 3]^C^CChange;\;P;C;3;;
P1_16[<-<-Change to Colour 4]^C^CChange;\;P;C;4;;
P1_17[--]
P1_18[&Kenny](alert "Kenny is Handsome")
P1_19[--]
P1_20[Image Menu]^C^C$I=TEST4.DOORS $I=*

***IMAGE
**DOORS
[DOORS Created by Kenny Ramage ]
[DOORS(D1,DOOR1)]INSERT;*D1;\;;
[DOORS(D2,DOOR2)]INSERT;*D2;\;;
[DOORS(D3,DOOR3)]INSERT;*D3;\;;
[DOORS(D4,DOOR4)]INSERT;*D4;\;;
[DOORS(D5,DOOR5)]INSERT;*D5;\;;
[DOORS(D6,DOOR6)]INSERT;*D6;\;;
[DOORS(D7,DOOR7)]INSERT;*D7;\;;
[DOORS(D8,DOOR8)]INSERT;*D8;\;;
[DOORS(DDOOR,DOUBLE DOOR)]INSERT;*DDOOR;\;;
[DOORS(HDDOOR,DOOR & HALF)]INSERT;*HDDOOR;\;;
[ FITTINGS]$I=KENNY.FITTINGS $I=*

***TOOLBARS
**TESTBAR
ID_1 [_Toolbar("Testbar", _Floating, _Show, 202, 163, 1)]
ID_2 [_Button("Test Tool 1", "ICON.bmp", "ICON_24_BLANK")]
```

```
^C^C(Alert "Test Tool 1")
ID_3 [_Button("Test Tool 2", "ICON0041.bmp", "ICON_24_BLANK")]
^C^C(Alert "Test Tool 2")
ID_4 [_Button("Test Tool 3", "ICON8467.bmp", "ICON_24_BLANK")]
^C^C(Alert "Test Tool 3")
```

Let's take a closer look at the toolbars section :

*****TOOLBARS is, of course, the label for the start of the toolbars section. **TESTBAR is the label for the Toolbar Subsection.**
The first Toolbar definition line (ID_1) defines the characteristics of the toolbar itself. This is made up of 6 parts :

tbarname	The string that names the toolbar. ("Testbar")
orient	The orientation of the toolbar. (Floating) Acceptable values are Floating, Top, Bottom, Left and Right.
visible	The visibility of the toolbar. (Show) Acceptable values are Show and Hide.
xval	The x co-ordinate of the toolbar (202) Measured from the left edge of the screen to the right side of the toolbar. (in pixels)
yval	The y co-ordinate of the toolbar. (163) Measured from the top edge of the screen to the top of the toolbar. (in pixels)
rows	Number of rows in the toolbar (1)

The second and remaining Toolbar definition lines (ID_2 to ID_4) define the characteristics of the buttons. They are each made up of 3 parts :

btnname	The string that names the button. ("Test Tool 1")
id_small	The name of the small image bitmap (ICON.bmp) (16 x 16 bitmap)
id_big	The name of the large image bitmap (ICON_24_BLANK) (24 x 24 bitmap)
macro	Action/Command assigned to the button.

Did you notice that a blank tile is in place of the large 24 x 24 bitmap.
The reason for this, of course, is because we never defined a large bitmap.
If I require large bitmaps I use an imaging editing package to increase the size of my existing 16 x 16 bitmap to 24 x 24 and save it as a large bitmap, adding "24" to the end of the name.

Anyway, that's about it. I hope that you managed to muddle your way through this lot and that you will soon be writing your own partial menu's.

I don't know about you, but this is thirsty work, so I'm off for an extremely large, cold

beer. (or 10). Cheers for now.....

Loading Partial Menu's

Fire up AutoCAD and follow these instructions :

- Select "Tools" "Customise Menu's" from pulldown.
- Select "Browse"
- Change file type to "Menu Template (*.MNU)"
- Open Directory where TEST.MNU resides.
- Select "TEST.MNU" then press "OK"
- Select "Load".
- Answer "Yes" when prompted.
- Click on "Menu Bar" tab.
- From "Menu Group" pulldown select "TEST".
- In the "Menu's" section, select "Test Menu".
- Press "Insert" to add it to the Menu Bar.
- Select "Close"

Un-Loading Partial Menu's

To Un-Load Partial Menu's do the following :

- Select "Tools" "Customise Menu's" from pulldown.
- In the Menu Groups list, highlight the menu that you wish to remove.
- Select "Unload".
- Select "Close".

The menu will be removed.

Menu files can also be loaded and un-loaded programmatically.
The methods of doing this will be covered later in the tutorial.

Just a couple of comments on Loading and Un-Loading menu's.
When loading a menu file, AutoCAD compiles the .MNU file into .MNC and .MNR files. The .MNC file is a compiled version of the .MNU file and the .MNR file contains the bitmaps used by the menu.
AutoCAD also generates a .MNS file. This is an ASCII file that is initially the same as the .MNU file but is modified by AutoCAD each time you make changes to the menu file through the interface. (such as modifying the contents of a toolbar.

***HINT* : If you have modified a menu file and it does not display correctly, or gives you other hassles, then delete the .MNC, .MNR and .MNS files. This forces the menu to re-compile.**

Creating Hatch Patterns

You want to what!!! Create a hatch pattern? Crikey, you're brave.
Honestly, though, simple hatch patterns are quite easy to create. It's the complicated ones that take time, effort, a good understanding of how to create hatches and linetypes, some knowledge of geometry and quite a bit of imagination.

Before proceeding with this tutorial, I would strongly recommend that you read my tutorial on creating custom linetypes. Linetypes are used extensively throughout hatch patterns and a good understanding is a requisite. You can find my tutorial on custom linetypes elsewhere in this manual.

Let's take a look at a simple hatch pattern first. The same principal applies to even the most complicated hatch pattern so, as my Mum often says, "Pay attention!!"

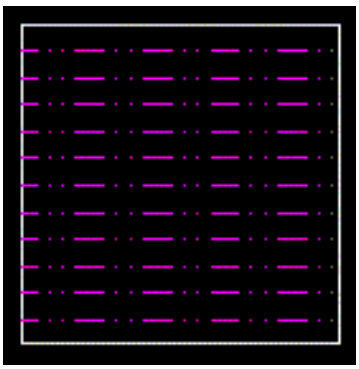
A hatch pattern definition is stored in a simple ASCII text file with an extension of PAT. You can append a hatch pattern to an existing file if you wish, or you can store it in it's own file. If you store it in it's own file, the file name must match the name of the hatch pattern. We are going to create our own hatch definition or pattern file.
O.K. Open Notepad and add this line :

***Dashdot, Dashes and dots**

This is the name of our hatch pattern followed by a description of the pattern, separated by a comma. (,)
Now add this on the next line :

0, 0,0, 0,0.5, 0.5,-0.25,0,-0.25,0,-0.25

This line describes a single drawing pass over the hatch pattern. In this example we are only using a single pass so we will only get one line. Save your file as "Dashdot.pat". If you loaded this hatch pattern and added it to your drawing, it would look like this :



The syntax of each drawing line is as follows :

Angle, X,Y Origin, Offset-x,Offset-y, Pen Command Pattern

Let's look a bit closer at each field :

0, 0,0, 0,0.5, 0.5,-0.25,0,-0.25,0,-0.25

The first field, Angle, which in our case is "0", determines the angle at which the line is to be drawn. In our case it will be drawn horizontally. Don't confuse this with the angle of the hatch pattern which is controlled by the AutoCAD Hatch command. Look at a hatch pattern as a successive series of lines that are drawn from left to right, then from down to up.

0, 0,0, 0,0.5, 0.5,-0.25,0,-0.25,0,-0.25

The second field is the X,Y Origin. This controls the starting point of the line segment. This is not an AutoCAD co-ordinate, but rather a relative distance from the current Snap base point of the drawing. All hatch patterns have a point of origin. Since this point of origin is the same throughout, you're assured that the patterns will line up.

0, 0,0, 0,0.5, 0.5,-0.25,0,-0.25,0,-0.25

The third field is the X-Offset and Y-offset values. 0 is the X-Offset and 0.5 is the Y-Offset. The hatch pattern will begin at an arbitrary origin and proceed to draw a group from left to right, then advance upward in the Y direction.
0,0.5 means that each successive line in the pattern will move to the right 0 units and up by 0.5 units. This results in the 0.5 spacing between the lines. The offset is relative to the initial angle given in the line, so that angle forms the X axis for the offset.
The Y value Offset is quite easy to understand - it gives you the spacing between the lines. But why would you want to offset the X value, and what effect does that have? Think of a brick wall. Each successive line of bricks is offset to the right a little to create a pattern. So, 0.5,1 would space the bricks upward by 1 unit, and every other line would be offset by 0.5 to the right of the origin, creating a "stepladder" effect.

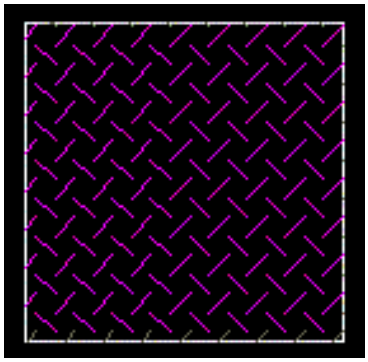
0, 0,0, 0,0.5, 0.5,-0.25,0,-0.25,0,-0.25

The fourth and final group is the linetype definition or the dash dot pattern. In words, this pattern is saying:
"Draw a line 0.5 units long, lift the pen for 0.25 units, draw a dot, lift the pen for 0.25 units, draw another dot, lift the pen for 0.25 units, draw a third dot, lift the pen for 0.25 units and then repeat the process".

The dashdot pattern was drawn using only one definition line, let's try one with two. Open a new file with Notepad and add this :

***VASTRAP, Vastrap Checkered Plate**
0, 0,0.09375, 0.25,0.25, 0.25,-0.25
90, 0.125,0.21875, 0.25,0.25, 0.25,-0.25

Close the file and save it as "Vastrap.pat". This hatch will produce a pattern like this :



Even though I defined the pattern with 0 and 90 degree lines, you can rotate the pattern to get the desired effect. I made the pattern at 0 and 90 degrees to avoid having to calculate the angles. (Chicken hey!)

As I said at the beginning, simple hatch patterns are quite easy to create, but the complicated one's? Well, that's another story.
Would you like a couple of hundred hatch patterns to play around with and analyze? Some simple, some "very" complicated! You would? Then you'll find them in the Zip file provided.

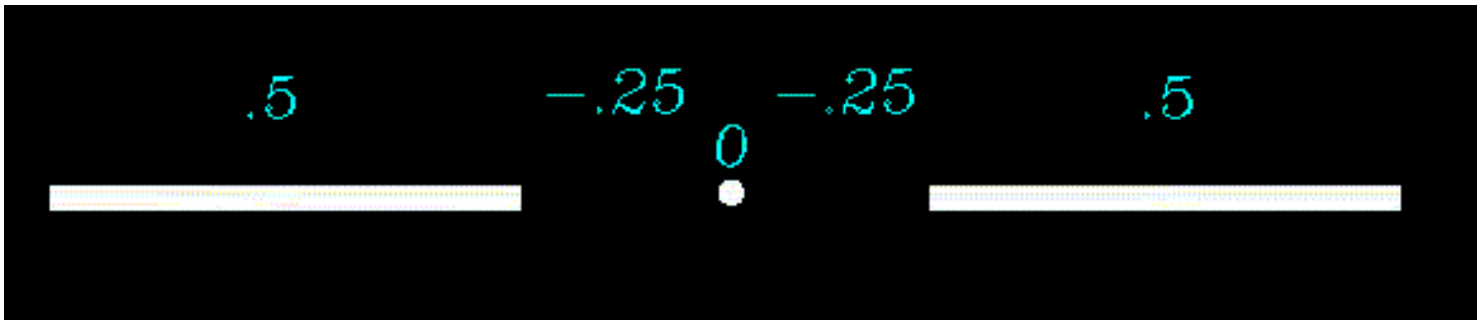
Custom Linetypes

There are two different ways of creating linetypes within AutoCAD. You can write your own linetype definition file, or you can use AutoCAD's built in function to create it for you. In this tutorial we'll look at both methods. Let's start with the AutoCAD built in function first.

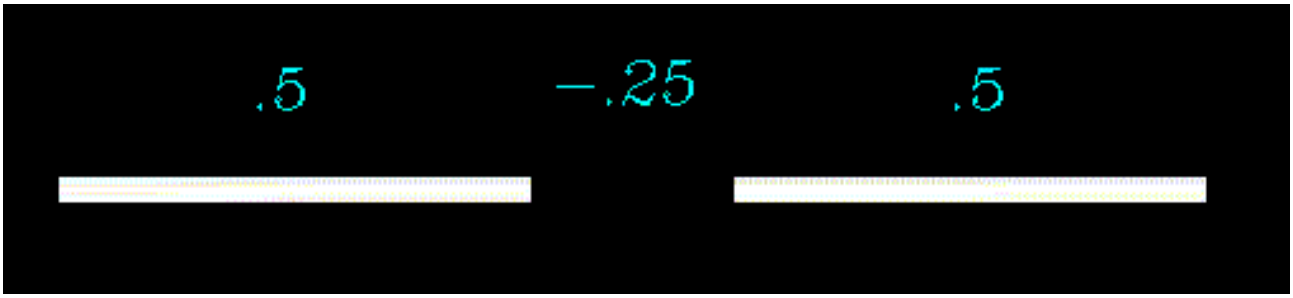
Before we start though, let's have a look at what a linetype is.

A linetype is a series of positive and negative numbers that tell a plotter how long to lower or raise a pen. Positive numbers lower the pen, and negative numbers raise it. The normal length of a dash is .5; a dot is 0.
e.g. 0,-.25 would produce a series of dots. The first 0 produces the dot; the -.25 raises the pen .25 of a unit. The series then repeats itself.

A dash-dot would be .5,-.25,0,-.25.



A dashed line would be .5,-.25.



Easy hey! Now let's have a look at creating out own custom linetype. Let's say we want a linetype to be dash, dot, dot, dot.

The format would be .5,-.25,0,-.25,0,-.25,0,-.25. and we'll call the linetype "dashdot3". Fire up AutoCAD and enter the following at the command line :

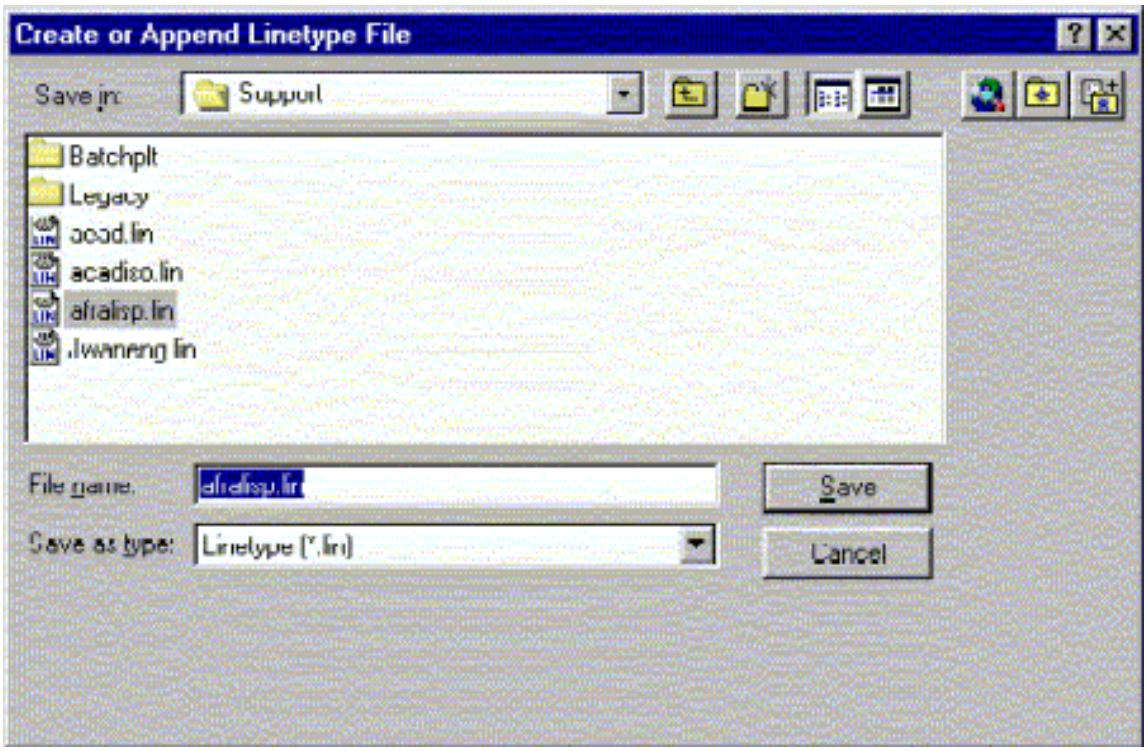
Command: *-linetype*

Current line type: "ByLayer"

Enter an option [?/Create/Load/Set]: *c*

Enter name of linetype to create: *dashdot3*

After pressing "Enter", the following dialog will appear :



Enter "*Afralisp*" as the file name and select "*Save*".

Creating new file

Descriptive text: *_ . . . _ . . . _ . . . _*

Enter linetype pattern (on next line):

A,.5, -.25, 0, -.25, 0, -.25, 0, -.25

New linetype definition saved to file.

Enter an option [?/Create/Load/Set]:

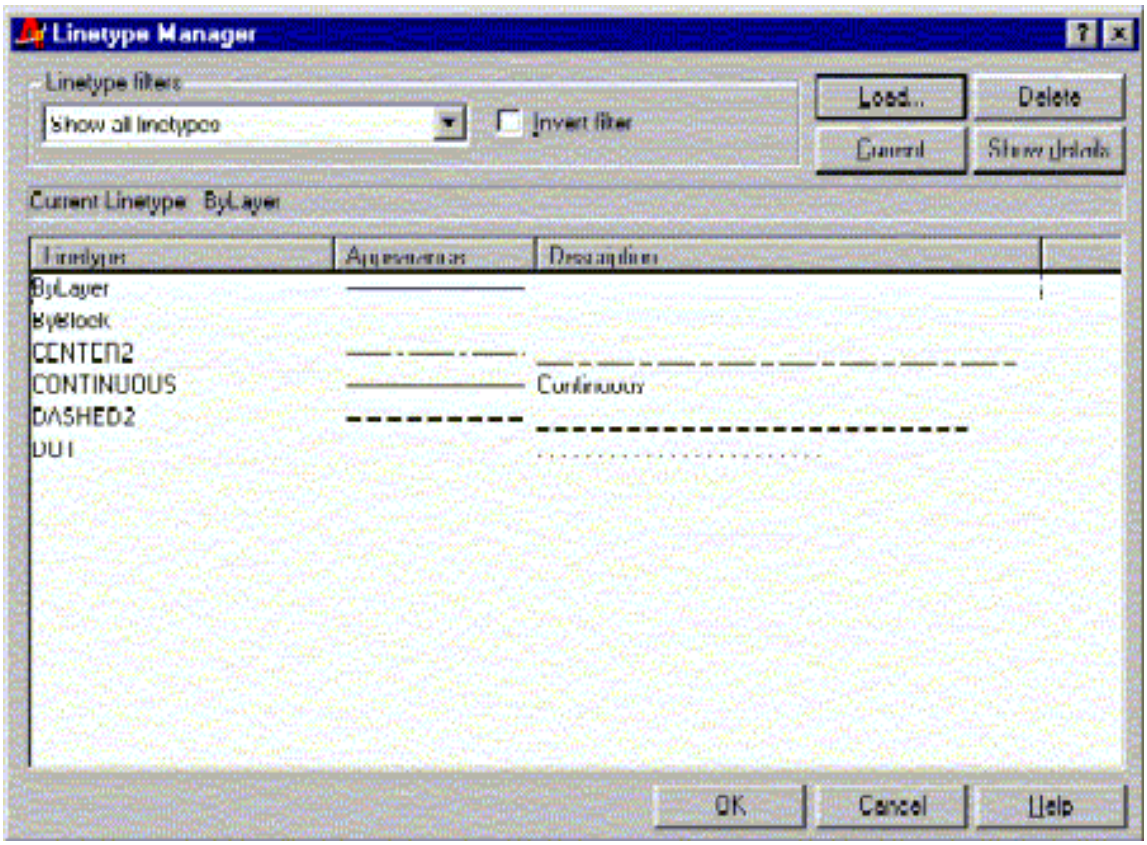
Select "*Enter*" to complete the process.

We could have appended the linetype definition to an existing file, but in this case we've created a new file.
If you want to change an existing linetype definition, just create a new linetype, giving it the same name as the old one. AutoCAD will ask you if you want to overwrite the existing

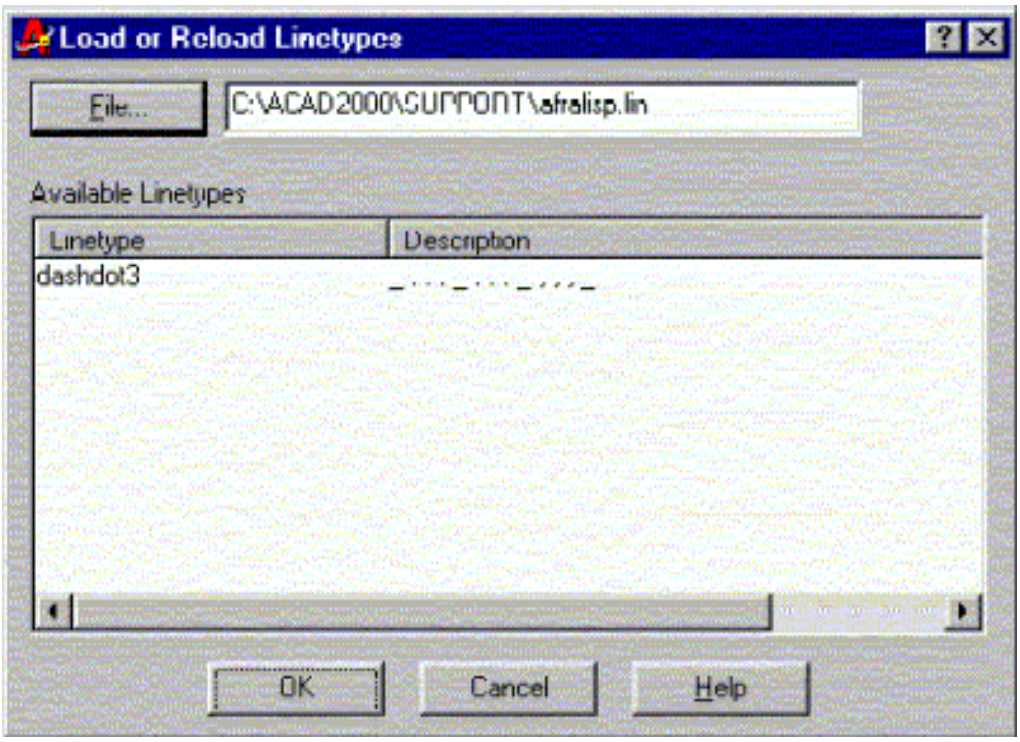
linetype; answer yes. You can then enter the new definition and description.

Note : Complex linetypes cannot be created using the "Linetype" command at the command line.

To load and set your new linetype, enter "Linetype" at the command line. The "Linetype Manager" dialogue will appear :

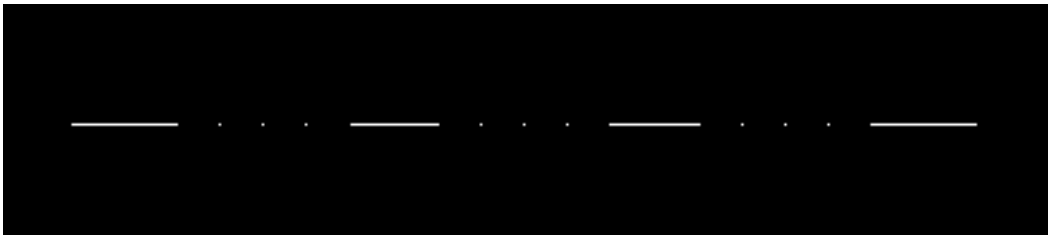


Select "Load". The Load dialog will appear :



Select "File" and then "AfraLisp.lin"
Next select "dashdot3" linetype and then "OK"

In the "Linetype Manager" dialogue, select "dashdot3" again and then "Current" followed by "OK".
Your "dashdot3" linetype should now be loaded and set. Draw a line. It should look like this :



To create your own linetype definition is very simple. First locate your newly created "Afralisp.lin" file, and open it using Notepad. It should look like this :

```
;
;AfraLisp Linetype definitions
;Written November 2001
;
*dashdot3,_ . . . _ . . . _ . . . _
A,0.5,-0.25,0,-0.25,0,-0.25,0,-0.25
```

Yours will not have the first two lines as I added them later. Just precede any comments you wish to add with ";".
As you can see, your linetype definition consists of 2 lines.
The first line consists of "*" followed by the linetype name followed by the description

separated by a comma ",".
The description is optional and can be a sequence of dots, spaces and dashes or a comment such as "A Dashed Line followed by 3 Dots". If you omit the description, do not put a comma after the linetype name. If you include a description, it should be no more than 47 characters long. (Why? Who knows!!)

The second line starts with the "alignment" field which you specify by entering "A". AutoCAD only supports one type of alignment field at this time so you will always enter "A".
Next comes a comma "," followed by your linetype definition.

Pretty easy hey. Next we'll have a look at complex linetypes.

The syntax for complex linetypes is similar to that of simple linetypes in that it is a comma-delimited list of pattern descriptions. The difference is, is that complex linetypes can include shape and text objects as pattern descriptors, as well as the normal dash-dot descriptors of simple linetypes.
In this tutorial we are going to look at creating a complex linetype using a text object as a descriptor.

The syntax for text object descriptors in a linetype definition is as follows :

["string" , stylename] or

["string" , stylename , transform]

That's great Kenny, but I haven't a clue what this all means!!!
O.K. Don't worry, we'll go through it step by step.
Open up your AfraLisp.lin file that you created earlier and add the following :

```
;
;AfraLisp Custom Linetypes
;Written November 2001
;
*dashdot3,_ . . . _ . . . _ , , , _
A,0.5,-0.25,0,-0.25,0,-0.25,0,-0.25
;
*AFRALISP,---- AL ---- AL ---- AL ---- AL
A,1.0,-0.25,["AL",STANDARD,S=1,R=0,X=0,Y=-0.25],-1.25
```

If you loaded and set this linetype, it would look like this :

If we put this into words, we would be saying, "Draw a line 1 unit long, lift the pen for 0.25 of a unit, then add "AL" as text using "STANDARD" font style, now lift the pen for 1.25 units, and then repeat the process".

Let's have a wee look at the "AFRALISP" linetype :

```
*AFRALISP,---- AL ---- AL ---- AL ---- AL
```

The first line, of course is the linetype name followed by the description.

```
A,1.0,-0.25,["AL",STANDARD,S=1,R=0,X=0,Y=-0.25],-1.25
```

The second line contains the linetype definition. Lets, zoom into this.

"A" is the alignment which is defaulted.
"1.0" and "-0.25" are pen commands as is "-1.25" at the end of the definition.
["AL",STANDARD,S=1,R=0,X=0,Y=-0.25] is the one that interests us.
The syntax of these fields are as follows :

***string* ("AL")**

The text to be used in the complex linetype.

***style* (STANDARD)**

The name of the text style to be elaborated. The specified text style must be included. If it is omitted, use the currently defined style.

***scale* (1)**

S=value. The scale of the style is used as a scale factor by which the style's height is multiplied. If the style's height is 0, the S=value alone is used as the scale.
Because the final height of the text is defined by both the S=value and the height assigned to the text style, you will achieve more predictable results by setting the text style height to 0. Additionally, it is recommended that you create separate text styles for text in complex linetypes to avoid conflicts with other text in your drawing.

***rotate* (0)**

R=value or A=value. R= signifies relative or tangential rotation with respect to the lines elaboration. A= signifies absolute rotation of the text with respect to the origin; all text has the same rotation regardless of its relative position to the line. The value can be appended with a d for degrees (if omitted, degree is the default), r for radians, or g for grads. If rotation is omitted, 0 relative rotation is used.

Rotation is centered between the baseline and the nominal cap heights box.

***xoffset* (0)**

X=value. This field specifies the shift of the text in the X axis of the linetype computed from the end of the linetype definition vertex. If xoffset is omitted or is 0, the text is elaborated by using the lower-left corner of the text as the offset. Include this field if you want a continuous line with text. This value is not scaled by the scale factor that is defined by S=.

***yoffset* (-0.25)**

Y=value. This field specifies the shift of the text in the Y axis of the linetype computed from the end of the linetype definition vertex. If yoffset is omitted or is 0, the text is elaborated by using the lower-left corner of the text as the offset. This value is not scaled by the scale factor that is defined by S=.

Well that's about it for complex linetypes. But before I go, here's a few more custom

linetypes that you can play around and practice with :

```
;
;
;AfraLisp Custom Linetypes
;Written November 2001
;
;
*dashdot3,_ . . . _ . . . _ , , , _
A,0.5,-0.25,0,-0.25,0,-0.25,0,-0.25
;
*AFRALISP,---- AL ---- AL ---- AL ---- AL
A,1.0,-0.25,["AL",STANDARD,S=1,R=0,X=0,Y=-0.25],-1.25
;
*DIESEL,---- DIESEL ---- DIESEL ---- DIESEL ---- DIESEL ---- DIESEL
A,50,-2,["DIESEL",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-6.5
;
*BEER,---- BEER ---- BEER ---- BEER ---- BEER ---- BEER
A,50,-2,["BEER",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-6.5
;
*DRINKING_WATER,---- DRINKING H20 ---- DRINKING H20 ---- DRINKING H20
A,50,-2,["DRINKING H20",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-14
;
*ELECTRICAL,---- ELECT ---- ELECT ---- ELECT ---- ELECT ---- ELECT
A,50,-2,["ELECT",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-5.5
;
*HIGH_TENSION,---- HT ---- HT ---- HT ---- HT ---- HT ---- HT ----
A,50,-2,["HT",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-2
;
*OIL,---- OIL ---- OIL ---- OIL ---- OIL ---- OIL ---- OIL ----
A,50,-2,["OIL",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-3
;
*OPTIC,---- OPTIC ---- OPTIC ---- OPTIC ---- OPTIC ---- OPTIC ----
A,50,-2,["OPTIC",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-5.5
;
*PHONE,---- PHONE ---- PHONE ---- PHONE ---- PHONE ---- PHONE ----
A,50,-2,["PHONE",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-6
;
*PLANT_WATER,---- PLANT H20 ---- PLANT H20 ---- PLANT H20
A,50,-2,["PLANT H20",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-11
;
*SECURITY_FENCE,---- X ---- X ---- X ---- X ---- X ---- X ----
A,50,-2,["X",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-1
;
*SEWER,---- SEWAGE ---- SEWAGE ---- SEWAGE ---- SEWAGE ----
A,50,-2,["SEWAGE",STANDARD,S=1.25,R=0.0,X=-1,Y=-.5],-7
```

O.K. Now I'll put you all out of your misery!!
Under the Express Tools pull down menu you will find the following :
"*Make Linetype*" and "*Make Shape*". (Sorry A2K only).
Now don't say I'm not nice to you!!

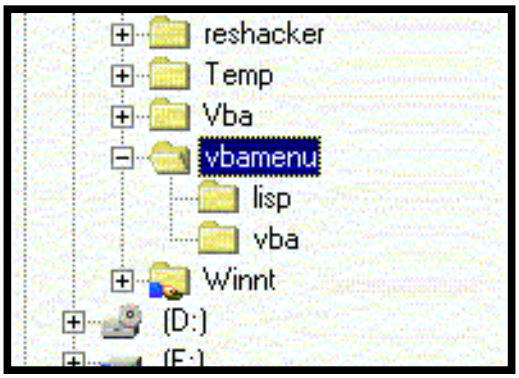
Menu Loader

Loading partial men files can be a real pain. Having to first add all the menu directories to the AutoCAD Support Path and then installing the menu files becomes a real hassle, especially if you've got a number of workstations to take care of.

Here's a little application that takes a lot of the pain out of multiple standard menu installation.
I have used a small partial menu "Vbamenu.mnu" that contains one pulldown menu and one toolbar as an example.

First, you need to have your support file directory structure set up and ready :

c:/vbamenu
c:/vbamen/lisp
c:/vbamenu/vba



In this example you can see that I've set up my main directory, "c:/vbamenu" with two sub-directories, "lisp" and "vba."
My menu and lisp files, "vbamenu," "vba.dll," and "vbamenu.lsp" are stored in "vbamenu," whilst I would store any additional support files in the other two sub-directories.

In essence, these are the directories that I would need to add to my AutoCAD support path.

Next, fire up AutoCAD and open a blank drawing.

Now type this at the command line :

(load "c:\\vbamenu\\vbamenu)

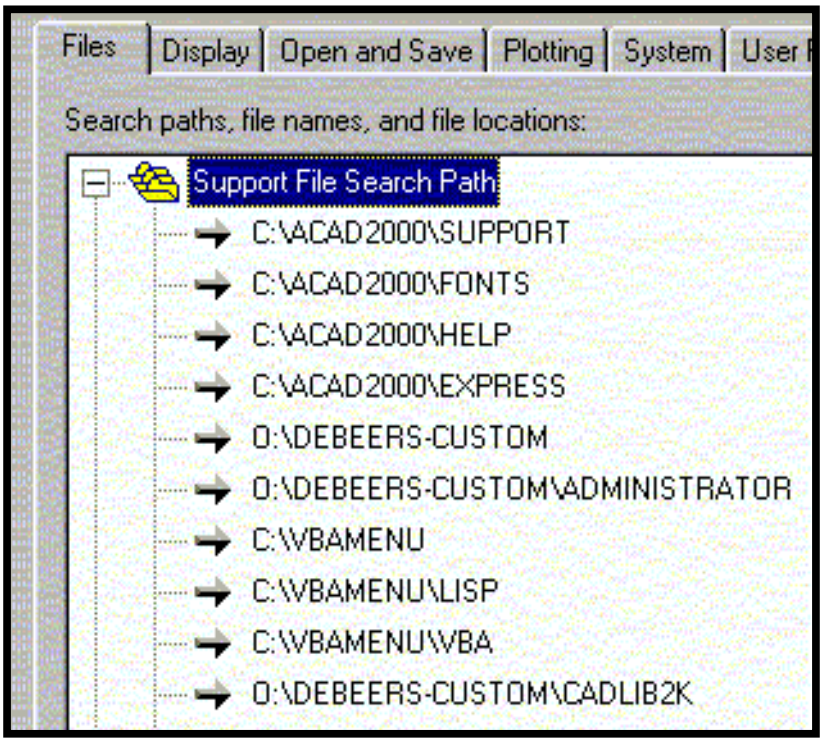
Then press "Enter"

A message should appear on the command line asking you to type *"vbamenu"* to run.

Do just that and stand back.

Voila, you should have a new pull down menu and a new toolbar.

Check out your support path. Three new directories should have been added:



Heres the coding :

```
;inform the user how to run the application
(prompt "\nVBA Menu Installer Loaded...Type VBAMENU to run.....")

;define the function
(defun C:VBAMENU ( / flag1 loaded temp)

        ;add the first support path
        (addSupportPath "C:\\VBAMENU" 6)

        ;add the second
        (addSupportPath "C:\\VBAMENU\\LISP" 7)

        ;add the third
        (addSupportPath "C:\\VBAMENU\\VBA" 8)

;set the flag
(setq flag1 T)

;check if the menu is not already loaded
(setq loaded (menugroup "VBAMENU"))

;if it is not loaded
(if (= loaded nil)
```

```
   ;do the following
   (progn

        ;find the menu file
        (setq temp (findfile "VBAMENU.MNU"))

           ;if you find the menu
           (if temp

                ;do the following
                (progn

                        ;switch off dialogues
                        (setvar "FILEDIA" 0)

                        ;load the menu
                        (command "menuload" "VBAMENU")

                        ;switch on dialogues
                        (setvar "FILEDIA" 1)

                        ;install the pulldown
                        (menucmd "P11=+VBAMENU.POP1")

                        ;inform the user
                        (prompt "\nLoading VBA Menu....\n")

                );progn

                ;menu not find, so do the following
                (progn

                        ;inform the user
                        (alert "Cannot Locate VBA Menu.
                         \nCall System Administrator.")

                        ;clear the flag
                        (setq flag1 nil)

                );progn

           );if

    );progn

);if

                ;if the flag is set
                (if flag1
```

```
                        ;inform the user
                        (prompt "\nVBA Menu Loaded....\n")

                );if

    (princ)

);defun



;subroutine to add support path
(defun addSupportPath (dir pos / tmp c lst)

        (setq tmp ""
              c -1
        )
        (if
                (not
                  (member (strcase dir)
                        (setq lst
                          (mapcar 'strcase
                                (parse (getenv "ACAD") ";")
                          )
                        );setq
                  );member
                );not
        (progn
                (if (not pos)
                        (setq tmp (strcat (getenv "ACAD") ";" dir))
                        (mapcar '(lambda (x)
                        (setq tmp (if (= (setq c (1+ c)) pos)
                                        (strcat tmp ";" dir ";" x)
                                        (strcat tmp ";" x)
                                  )
                        )
                        )
                lst
                )
        )
        (setenv "ACAD" tmp)
        )
        )
(princ)
)

;parsing routine
(defun parse (str delim / lst pos)
        (setq pos (vl-string-search delim str))
        (while pos
```

```
                (setq lst (cons (substr str 1 pos) lst)
                        str (substr str (+ pos 2))
                        pos (vl-string-search delim str)
                )
        )
        (if (> (strlen str) 0)
                (setq lst (cons str lst))
        )
                (reverse lst)
)

(princ);clean load
```

A big thank you to ActiveX.com for allowing me to borrow a wee bit of their coding.

To make sure that your menu is loaded every time AutoCAD starts, place this coding in your Acad.Lsp file :

```
(defun-q VBAMENUSTARTUP ()

(setq flag1 T)

(setq loaded (menugroup "VBAMENU"))

(if (= loaded nil)

   (progn

        (setq temp (findfile "VBAMENU.MNU"))

           (if temp

                (progn

                        (setvar "FILEDIA" 0)

                        (command "menuload" "VBAMENU")

                        (setvar "FILEDIA" 1)

                        (menucmd "P11=+VBAMENU.POP1")

                        (prompt "\nLoading VBA Menu....\n")

                );progn

                (progn

                        (alert "Cannot Locate VBA Menu.
                         \nCall System Administrator.")
```

```
                        (setq flag1 nil)
                );progn
            );if
    );progn
);if
                (if flag1
                        (prompt "\nVBA Menu Loaded....\n")
                );if
    (princ)

);defun-q
(setq S::STARTUP (append S::STARTUP VBAMENUSTARTUP))
```

Define Function - (defun).

We've all seen something like this :

(defun C:DDSTEEL (/ t1 t2 h1 h2 ang)

This, of course, is the first line of an AutoLISP routine.
But, do you know what it all means?
When I started writing AutoLISP code, I was totally confused!!
The more I read, the more in a muddle I got.
Global, Local, Argument, Define, Aghhh.....
So, if you are as confused as I was, read on.

Let's start at the very beginning. (A very good place to start, say you!)
The name of a program, or function, must be defined in the first statement, which is done by using the command :

(defun *[Define Function]*

Why is there not a closing parenthesis after the (defun command?
There is! In fact the last parenthesis in the program is the one that closes (defun.

Anyway, let's carry on.
After (defun must come the name of the program :

(defun DDSTEEL

You do realise that the name of the AutoLISP file, is not necessarily the name of the program. In fact, one AutoLISP file can have several programs inside.

Hey, that was easy, I hear you say. Now comes the hard part!!
After you've named your program you have three choices.

First, do nothing by using ( ) :

(defun DDSTEEL ( )

What you are saying here is that every variable that you use in your function is GLOBAL. A GLOBAL variable is one that doesn't lose it's value when the program ends. For example, if PT3 was defined as 45.2 in your program, it would still have the value of 45.2 when your program ends and would retain that value until you replace it with another value or, start a new drawing.

Secondly, you can declare your variables as LOCAL.
To do this you precede your variables with / :

(defun DDSTEEL (/ p1 p2 p3)

The following example shows the use of LOCAL symbols :

```
(defun TEST ( / ANG1 ANG2)

     (setq ANG1 "Monday")
     (setq ANG2 "Tuesday")

     (princ (strcat "\nANG1 has the value " ANG1))
     (princ (strcat "\nANG2 has the value " ANG2))
   (princ)
);defun
```

Now, at the AutoCAD command line, assign the variables ANG1 and ANG2 to values other than those used by the TEST function :

Command: (setq ANG1 45)
Command: (setq ANG2 90)

Verify their values :

Command: !ANG1 [Should return 45.0]
Command: !ANG2 [Should return 90.0]

Now run our function :

Command: (load "TEST")
Command: (TEST)

The function should return :

ANG1 has the value Monday
ANG2 has the value Tuesday

Now check the current values of ANG1 and ANG2 :

Command: !ANG1
Command: !ANG2

You will find that they still have the values of 45.0. and 90.0 respectively.

A LOCAL variable is one that has a value only for that program while the program is running.

The third option is to list a variable without the /.
This means that a variable is set up to receive a value passed to it from outside the

program. eg :

```
(defun DTR (a)
  (* PI (/a 180.0))
)
```

To use this, we must pass the value of the argument to the function :

(DTR 90.0)

Here is another example :

Let us write a function that calculates the area of a circle.
The formulae for calculating the area of a circle is:

Area = PI x radius squared or, PI x r x r

Our program would look like this :

```
(defun acirc (r)
   (setq a (* (* r r) PI))
   (setq a (rtos a))
   (princ (strcat "\nArea = " a))
 (princ)
)
```

Now try it :

Command: (load "acirc")
Command: (acirc 24)

It should return :

Area = 1809.6

You can, of course combine all three options, for example :

(defun DDSTEEL (a / b c d)

DDSTEEL is the name of the function. Variable a is an argument and receives the first value passed to it from outside the program.

Variables b, c, and d are all locals and lose their values once the program has ended.

Another option that can be used with (defun is if the name of the function is preceded with C:

(defun C:DDSTEEL ()

Because of the C: you don't have to use parenthesis to call the function.
AutoCAD now thinks of that function as an AutoCAD command.
This is only true if you call the function from the AutoCAD command line.
If you call the function from within another function you must precede it with C:

(C:DDSTEEL)

It's a good idea to leave all variables as global whilst you are writing your program so that you can check their values during debugging.

Program Looping.

**AutoLisp uses 2 kinds of loops, namely (repeat) and (while).
Let's have a look at the (repeat) function first :**

(repeat).

The (repeat) function is a simple looping structure. It executes any number of statements a specific number of times. Like the (progn) function, all of it's expressions get evaluated, but they get evaluated once each loop. Here's a simple example :

```
(defun c:loop ()
        (setq pt (getpoint "\nCentre of Rotation : "))
        (setq n  (getint "\nEnter Number of Steps : "))

        (repeat n
                (command "Rotate" "L" "" pt "20")
        )
   (princ)
)
```

**Now draw a circle anywhere on the screen and then run the routine.
The circle should move around.
Note that the variable that controls the number of loops must be an integer.**

(while).

The (while) function loops like (repeat) except that (while) has a conditional test. (while) will continue looping through a series of statements until the condition is nil. Here's an example :

```
(defun c:loop1 ()
        (while
                (setq pt (getpoint "\nChoose a point : "))
                (command "point" pt)
        )
  (princ)
)
```

In this example, you can continue to pick points until you press Enter. (AutoLisp treats Enter as nil). When you press enter the loop will terminate.

Here's another example :

```
(defun c:loop2 ()
        (setq ptlist nil)
        (while
                (setq pt (getpoint "\nEnter Point or RETURN when done: "))
                (setq ptlist (append ptlist (list pt)))
        )
  (princ)
)
```

This example keeps on asking for a point and adding the point to a list of points, called ptlist. It uses the (append) function to merge the new point list to ptlist. As soon as you hit Enter the loop stops.
Run the routine, choose a few points and check the value of ptlist.
It should contain a long list of points.

The (while) function can also be used for programme iteration.
This means that a loop is continued until the results of one or more expressions, calculated within the loop, determine whether the loop is terminated. A common use of iteration is to increment a counter.
Have look at this example :

```
(defun c:loop3 ()
        (setq count 0)
        (while (< count 10)
                (princ count)
                (setq count (1+ count))
        )
  (princ)
)
```

You should get :

```
012345678910
```

If you know the number of times you want to loop, use (repeat), a much simpler command than (while).

Hint : Have you ever wondered how to make an AutoLisp routine Auto-Repeat?
Enclose the whole function or sub-function in a (while) loop.
This way, the function will keep on repeating until Enter or Cancel is hit.

Enough for now, my brain hurts.....

In fact, I think I'm going "Loopy-Loo"
Cheers....

Conditionals.

(If) is probably the most important and widely use condition statement. Unlike other languages though, you can match only one (if) statement with a *then* statement. The syntax is as follows :

```
(if xyz
        (then do this)
        (else do this)
)
```

Let's look at a simple example :

```
(defun c:testif ()
        (setq a (getreal "\nEnter a Number : ")
              b (getreal "\nEnter Second Number : ")
        );setq
        (if (= a b)
                (prompt "\nBoth Numbers are equal")
                (prompt "\nBoth numbers are not equal")
        );if
   (princ)
);defun
(princ)
```

If you need to evaluate more than one *then,* or *else* statement, you must use the (progn) function. Here's another example :

```
(defun c:testprogn ()
        (setq a (getreal "\nEnter a Number : ")
              b (getreal "\nEnter Second Number : ")
        );setq
        (if (= a b)
              (progn
                (prompt "\nBoth Numbers are equal")
                (prompt "\nHere is Another statement")
                (prompt "\nAnd Another One")
              );progn
                (prompt "\nBoth numbers are not equal")
        );if
   (princ)
);defun
(princ)
```

You can use as many statements as you like within the (progn) function.

You can also use (if) along with logical operators. They are functions that determine how two or more items are compared. The available logical operators are :

AND OR NOT

AND returns true if all arguments are true.
OR returns true if any of the arguments are true.
NOT returns true if it's argument is false and returns false if it's argument is true. Let's look at some examples :

```
(defun c:testand ()
        (setq a (getreal "\nEnter a Number : "))
        (if
            (and
                (>= a 5.0)
                (<= a 10.0)
            );and
                (prompt "\nNumber is between 5 and 10")
                (prompt "\nNumber is less than 5 or greater than 10")
        );if
   (princ)
);defun
(princ)
```

```
(defun c:testor ()
        (setq a (getstring "\nAre you Male? Y/N : "))
        (if
            (or
                (= a "y")
                (= a "Y")
            );or
                (prompt "\nHello Sir")
                (prompt "\nHello Madam")
        );if
    (princ)
);defun
(princ)
```

A Relation Operator is a function that evaluates the relationship between two or more items. Relationship Operators available are :

```
<       less than
>       greater than
<=      less than or equal to
>=      greater than or equal to
=       equal to
/=      not equal to
eq      are two expressions identical
equal   are two expressions equal
```

Let's look a bit closer at the (eq) and the (equal) functions.
The (eq) function determines whether two expressions are bound to the same object.

```
(setq a '(x y z))
(setq b '(x y z))
(setq c b)

(eq a c) would return nil, a and c are not the same list.
(eq c b) would return true, b and c are exactly the same list.
```

The (equal) function determines whether two expressions evaluate to the same thing. You can use the optional numeric argument, *fuzz,* to specify the maximum amount by which both expressions can differ and still be considered equal.

```
(setq a '(x y z))
(setq b '(x y z))
(setq c b)
(setq m 1.123456))
(setq n 1.123457))

(equal a c) would return true.
(equal c b) would return true.
(equal m n) would return nil.
(equal m n 0.000001) would return true.
```

What about a Multiple (if) function. The (cond) function works very much like (if), except (cond) can evaluate any number of test conditions.
Once (cond) finds the first condition that is true, it processes the statements associated with that condition. (It only processes the first true condition). Here's an example :

```
(defun c:testcond ()
```

```
        (setq a
           (strcase (getstring "\nSize of Bolt (M10,M12,M16): ")
           );strcase
        );setq
        (cond
                ((= a "M10") (prompt "\nYou have choosen M10"))
                ((= a "M12") (prompt "\nYou have choosen M12"))
                ((= a "M16") (prompt "\nYou have choosen M16"))
                (T (prompt "\nUnknown Bolt Size"))
        );cond
   (princ)
);defun
(princ)
```

The (cond) function takes any number of *lists* as it's arguments.
Each argument must be a list containing a test followed by any number of expressions to be evaluated.

Error Trapping.

This was another area that caused me quite a bit of confusion when I was first learning AutoLISP. All AutoLISP routines should contain Error Trapping. There is nothing worse than running a routine and finding out that once it has run, or been cancelled, your system variables, etc. have all changed and nothing works the same any more.

I am going to look at two types of Error Trapping here. Firstly, an Error Trapping routine build into a function, and secondly, a Global Error Trapping function. But first, some background on the AutoCAD *error* function.

AutoLISP provides a method for dealing with user (or program errors).
It is one of the only AutoLISP functions that is user-definable.
This is the *error* function.

(*error* *string*)

It is executed as a function whenever an AutoLISP error condition exists.
If there is an error, AutoCAD passes one argument to *error*, which is a string containing a description of the error.

The following function does the same thing that the AutoLISP standard error handler does. Print error and the description.

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

Before designing an Error Trapping function there is a couple of things to keep in mind.
First, the Error-Trap function must be called *error*. It also must have an argument passing variable. Our variable is called *errmsg*.
You can call this variable anything you like. Now, you can put anything you like in the body of this function. For example :

```
(defun *error* (errmsg)
 (princ "\nAn error has occurred in the programme. ")
 (terpri)
 (prompt errmsg)
 (princ)
)
```

To test this error trap, create a lisp file with the preceding code and load the file. Begin running any program or command that you like. In the middle of the program hit ESC or Ctrl C. (AutoCAD thinks that this is an error.)
Control should be passed to the Error Trap.

It is important to note, and courteous, that you should never change a users settings, including an existing Error Trap, without first saving it as a variable and then replacing it at the end of the program.
Remember, as well, to place the replacement of this Error Trap in your Error Trap routine as your program could also crash.

Here is an example of Error Trap build into an AutoLISP routine :

```
(defun c:drawline ()                                    ;define function
```

```
        (setq temperr *error*)                          ;store *error*
        (setq *error* trap1)                            ;re-assign *error*
        (setq oldecho (getvar "cmdecho"))               ;store variables
        (setq oldlayer (getvar "clayer"))
        (setq oldsnap (getvar "osmode"))
        (setvar "cmdecho" 0)                            ;set variables
        (setvar "osmode" 32)
        (command "undo" "m")                            ;undo mark
        (setq pt1 (getpoint "\nPick First Point: "))    ;get points
        (setq pt2 (getpoint pt1 "\nPick Second Point: "))
        (command "LAYER" "M" "2" "")                    ;change layer
        (command "Line" pt1 pt2 "")                     ;draw line
        (setq pt3 (getpoint pt2 "\nPick Third Point: "));get 3rd point
        (setvar "osmode" 0)                             ;switch off snap
        (command "Line" pt2 pt3 "")                     ;draw line
        (setvar "clayer" oldlayer)                      ;reset variables
        (setvar "osmode" oldsnap)
        (setvar "cmdecho" oldecho)
        (setq *error* temperr)                          ;restore *error*
   (princ)
)

(defun trap1 (errmsg)                                   ;define function
        (command "u" "b")                               ;undo back
        (setvar "osmode" oldsnap)                       ;restore variables
        (setvar "clayer" oldlayer)
        (setvar "cmdecho" oldecho)
        (setq *error* temperr)                          ;restore *error*
        (prompt "\nResetting System Variables ")        ;inform user
   (princ)
)
```

This routine simply asks for 3 points then draws a line, on layer 2, between them. As you can see, the existing (*error*) error trap is saved to the variable temperr. *error* is then re-assigned to the error trap, called trap1.
Any system variables such as object snaps and command echo, are saved as well as the current layer. An UNDO MARK is then put in place.
When an error occurs, the error trap first performs an UNDO BACK before resetting the drawing back to it's original settings.
Try choosing the first two points and then hitting ESC or Ctrl C.
Did you see what happened? The first line that was drawn was erased and your settings have been returned to their initial state.

The following is an example of an Error Trap using a Global Function :

Our *drawline* routine with some differences!!

```
(defun c:drawline ()                                    ;define function
        (initerr)                                       ;intit error
        (setvar "cmdecho" 0)                            ;reset variables
        (setvar "osmode" 32)
        (command "undo" "m")                            ;set mark
        (setq pt1 (getpoint "\nPick First Point: "))    ;get points
        (setq pt2 (getpoint pt1 "\nPick Second Point: "))
        (command "LAYER" "M" "2" "")                    ;change layer
        (command "Line" pt1 pt2 "")                     ;draw line
        (setq pt3 (getpoint pt2 "\nPick Third Point: "));get 3rd point
        (setvar "osmode" 0)                             ;reset snap
```

```
        (command "Line" pt2 pt3 "")                    ;draw line
        (reset)                                        ;reset variables
   (princ)
)

(princ)
```

Now our Global Error Trap named Error.Lsp

```
(defun error()                                         ;load function
(prompt "\nGlobal Error Trap Loaded")                  ;inform user
(princ)
);defun
;;;*=========================================================
(defun initerr ()                                      ;init error
  (setq oldlayer (getvar "clayer"))                    ;save settings
  (setq oldsnap (getvar "osmode"))
  (setq oldpick (getvar "pickbox"))
  (setq temperr *error*)                               ;save *error*
  (setq *error* trap)                                  ;reassign *error*
  (princ)
);defun
;;;*=========================================================
(defun trap (errmsg)                                   ;define trap
  (command nil nil nil)
  (if (not (member errmsg '("console break" "Function Cancelled"))
      )
    (princ (strcat "\nError: " errmsg))                ;print message
  )
  (command "undo" "b")                                 ;undo back
  (setvar "clayer" oldlayer)                           ;reset settings
  (setvar "blipmode" 1)
  (setvar "menuecho" 0)
  (setvar "highlight" 1)
  (setvar "osmode" oldsnap)
  (setvar "pickbox" oldpick)
  (princ "\nError Resetting Enviroment ")              ;inform user
  (terpri)
  (setq *error* temperr)                               ;restore *error*
  (princ)
);defun
;;;*=========================================================
(defun reset ()                                        ;define reset
  (setq *error* temperr)                               ;restore *error*
  (setvar "clayer" oldlayer)                           ;reset settings
  (setvar "blipmode" 1)
  (setvar "menuecho" 0)
  (setvar "highlight" 1)
  (setvar "osmode" oldsnap)
  (setvar "pickbox" oldpick)
  (princ)
);defun
;;;*=========================================================
(princ)
```

To run and test this you must load Error.Lsp before you load Drawline.Lsp.
As you can see, by using a Global Error routine you can save yourself the bother of writing individual error traps for each of your programs.

Error.Lsp could easily be loaded from your Acad.Lsp and would then be available whenever one of your routines wishes to call upon it.

Happy Error Trapping!!!!

Calculating Points - Polar

The Polar function is defined in the AutoCAD Customization manual as follows :

POLAR : Returns the UCS 3D point at a specified angle and distance from a point.

(polar pt ang dist)

This, I believe, is one of the most useful functions in the AutoCAD stable.
In a nutshell, you feed it a point, tell it the angle and distance from that point that you want to be, and it will return the second point.
Can you imagine having to do that using car, cadr, etc.
First you would have to break the list down into each separate component, do all the calculations on each individual item, and then re-construct the list.
What a pain!!

Following is an example of how to use POLAR to construct a simple square or rectangle from values input by the user :

```
(defun DTR (a)                                  ;degrees to radians function
(* PI (/ a 180.0))
);defun
;========================================
(defun C:BOX1 (/ IP P1 P2 P3 LENGTH HEIGHT      ;define function and declare
                 OLDSNAP OLDBLIP OLDLIGHT)      ;variables as local

(setq OLDSNAP (getvar "OSMODE")                 ;store system variables
      OLDBLIP (getvar "BLIPMODE")
     OLDLIGHT (getvar "HIGHLIGHT")
);setq
;========================================
(setvar "CMDECHO" 0)                            ;change system variables
(setvar "BLIPMODE" 0)
(setq IP (getpoint "\nInsertion Point: "))      ;get insertion point
(setvar "OSMODE" 0)                             ;switch off snap
(setq LENGTH (getreal "\nEnter Length: ")       ;get length of box
      HEIGHT (getreal "\nEnter Height: ")       ;get height of box
);setq
;========================================
(setq P1 (polar IP (DTR 0.0) LENGTH)            ;calculate first corner
      P2 (polar P1 (DTR 90.0) HEIGHT)           ;calculate second corner
      P3 (polar P2 (DTR 180.0) LENGTH)          ;calculate third corner
);setq
;========================================
(command "PLINE" IP "W" "" ""
         P1 P2 P3 "C"
);command                                       ;draw the box
;========================================
(prompt "\nRotation Angle: ")                   ;prompt the user for rotation
(command "ROTATE" "LAST" "" IP pause)           ;rotate the box
```

```
;=======================================
(setvar "OSMODE" OLDSNAP)                    ;reset system variables
(setvar "BLIPMODE" OLDBLIP)
(setvar "HIGHLIGHT" OLDLIGHT)
(princ)                                      ;exit quietly
);defun
```

As you can see, we first need to write a function to convert radians to degrees. This is because when we deal with angles in AutoLISP they must be in Radians.

Hint : This could be a Global function loaded from your Acad Lisp file.

Now to our main routine.

The first thing that we do is define the function and declare all the variables as local. (Only used within this program.)
Then we save certain system variables, before changing them, so that we can reset them later.
Next we ask the user for Insertion Point, Length and Height of the box.
We then use the POLAR function to calculate the remaining 3 corners of the box. (PT1, PT2 and PT3).
Then, just to be kind, we allow the user to rotate the box.
(That's why we drew the box using a Pline.)
Lastly, we reset all the system variables that we changed back to their original state.

Locating Files.

AutoLISP has two functions available to help us locate files.
They are the (findfile) function and the (getfiled) function.

(findfile).

The (findfile) function will only search the current AutoCAD search path if a drive/directory prefix is not supplied.
If the file is found, the full directory path is returned.

If no file is found, (findfile) returns nil.
The syntax of this function is as follows :

```
(findfile "filename")
```

Say you were looking for the file ACADR14.LSP.

```
(findfile "ACADR14.LSP)
```

Would return :

```
"C:\\ACADR14\\SUPPORT\\ACADR14.LSP"
```

Note :
AutoLisp allows you to use / or \\ for directory descriptors.

(getfiled).

The (getfiled) function will prompt the user for a file name using the standard AutoCAD file dialogue box. It will then return the file name either with the full path name or with the path name stripped.
The syntax of the (getfiled) function is as follows :

```
(getfiled "Title" "Directory Path and/or File name" "File Extension" Flag)
```

The "Title" argument is simply the name that will appear in the Title Bar of the dialogue box.

The "Directory Path and/or File Name" argument is the default directory path that the dialogue box will use. If a file name is included this name will appear in the File Name edit box. This can be null.

The "File Extension" function is the default file name extension. If it is null, it defaults to * (all file types). If the file type "dwg" is part of the "File Extension", an image preview is displayed.

There are four possible flags and they make use of the "sum of the flags" concept.

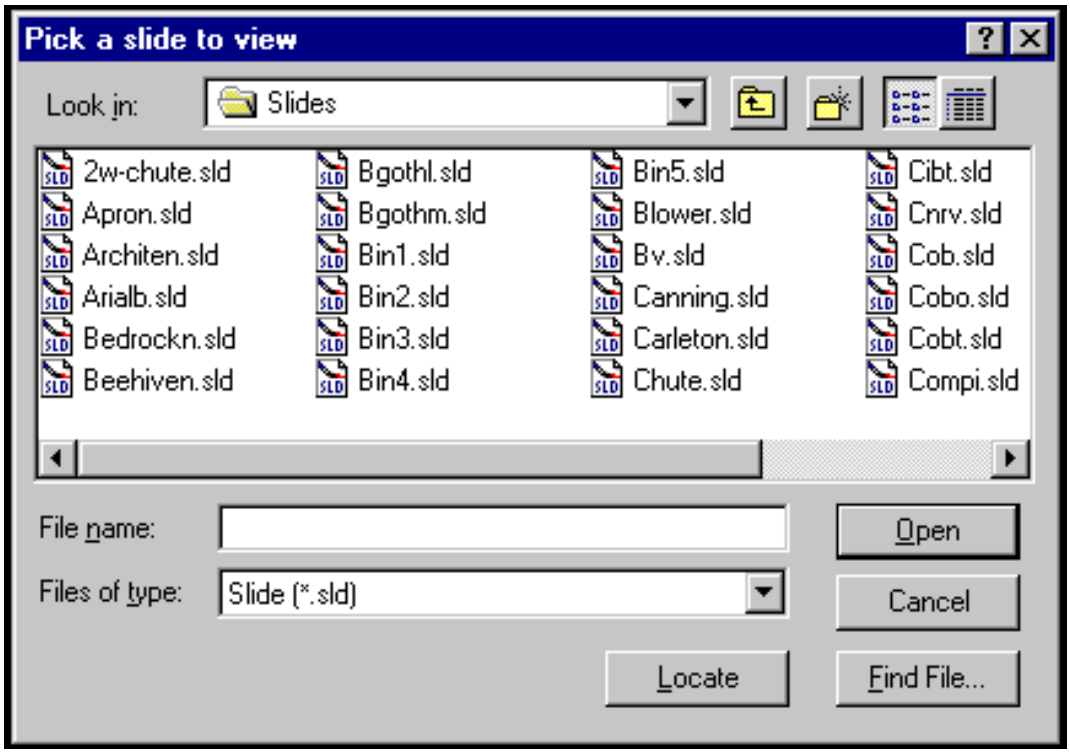
To combine any of the flags, just add them together. The flags are :

```
Flag 1. If you set this flag, the function will prompt for the name of a
        NEW file to create.
Flag 4. This flag will disable the "Type it" button. The user must then
        pick from the listed files rather than type a file name.
        If this flag is not set and the user selects the "Type it" button,
        the dialogue box disappears and (getfiled) returns a value of 1.
Flag 3. This flag allows the user to enter a file extension.
        If the flag is not set, only the extension used in the extension
        edit box will be used.
        The extension is also added to the file name if the user does not
        enter it in the File Name edit box.
Flag 8. If this flag is set and flag 1 is not set, (getfiled) searches in
        accordance to the AutoCAD library search path. It also strips the
        path and only returns the file name.
        If it is not set, it returns the entire path name.
```

Let's look at an example. We want to open a directory on c:/ called "Slides" containing a list of file that we would like to view. Our routine would look like this :

```
(defun c:slv ( / sl)
        (setq sl (getfiled "Pick a slide to view"
                           "C:/SLIDES/"
                           "sld"
                           10
                 );getfiled
        );setq
        (command "vslide" sl)
   (princ)
);defun
(princ)
```

Your dialogue box should look like this :



Take note of a couple of things.
See how it defaults to the C:/SLIDES directory;
The "Type it" button has been disabled; (Flag 2 was set.)
The full path name of the file was returned. (Flag 8 was set)
This is because C:/SLIDES is not in my AutoCAD search path.

As you can see, this is quite a useful function in that it can restrict your user to only certain directories and file types whilst still leaving them some flexiblity in their choice.

File Handling.

AutoLisp can only deal with one type of external file, namely an ASCII text file.
As well as this, AutoLisp can only read a file in sequential order. (It does not have random access.)
Despite these drawbacks, AutoLisp has certain basic tools that allow you to read and write one character at a time, or a full line at a time.
You can also append data to an existing file.

Working with external files is very simple.
First you "Open" the file.
Then you process the file by "Reading", "Writing" or "Appending" to it.
You then "Close" the file.
When you "Open" a file, Autolisp returns a file handle. This file handle is a name supplied by the operating system that let's you inform AutoLisp which file you want to work with.

Let's look at some examples.

```
(setq file (open "Testfile.txt" "w"))
```

AutoLisp should return something like this :

```
File: #28a27d2
```

This is the file handle returned by the operating system and stored in variable "file".
Try this now :

```
(write-line "This is some test data." file)
```

This "writes" a line of data to the file with handle "file".

```
(write-line "This is some more test data." file)
```

Let's write some characters to the same file :

```
(write-char 79 file)
```

This would write the ASCII character "O" to the file.

```
(write-char 75 file)
```

This would write the letter "K"
Now let's close the file :

```
(close file)
```

To read a file is just as simple :

```
(setq file (open "testfile.txt" "r"))
```

Open "Testfile.txt" to "Read".

```
(read-line file)
```

Read the first line.
Lisp should return :

```
"This is some test data "
```

Read the next line :

```
(read-line file)
```

Lisp should return :

```
"This is some more test data."
```

Let's read a character :

```
(read-char file)
```

Lisp will return :

```
79
```

It has returned the ASCII number.

```
(chr (read-char file))
```

Read the character AND convert it.
Lisp should return :

```
"K"
```

```
(read-line file)
```

Lisp should return "nil" as we have reached the end of the file.

Before moving on, you should always make sure that you close your files.

```
(close file)
```

Append is very similar to writing to a file except the file must already exist if you want to append to it.

**There are three other functions that write to an external file.
They are (princ), (prin1) and (print).
Let's have a look at them :**

```
(setq file (open "afile.txt" "w"))
(princ "This is the (princ) function" file)
(prin1 "This is the (prin1) function" file)
(print "This is the (print) function" file)
(close file)
```

Open the file "afile.txt". You should have the following :

```
This is the (princ) function"This is the (prin1) function"
"This is the (print) function"
```

All 3 functions display the result at the prompt line and send the output to a file. Here are the differences :

**(princ) displays strings without quotation marks.
(prin1) displays strings with quotation marks.
(print) displays strings with quotation marks and places a blank line before the expression and a space afterwards.**

Now we will look at a practical example :

The following is a Drawing Log Routine that logs the date, time, & Drawing Name of each Drawing Session. It writes a report to an ASCII Text file (Log.Txt).

```
(defun C:LOGIN ( / a c d file fp)
   (setq file (findfile "LOG.TXT"))
   (if (not file)
        (open "LOG.TXT" "w")
   );if
   (setq a (TODAY)
```

```
      TIME1 (TIME)
      c (getvar "DWGNAME")
      d (strcat "Drg Start   " a "  -  " TIME1 "  -  " c)
   );setq
   (if (/= c "Drawing.dwg")
      (progn
         (setq file (findfile "LOG.TXT")
            fp (open file "a")
         );setq
         (princ d fp)
         (princ "\n" fp)
         (close fp)
         (princ (strcat "\nLogged in at : " TIME1))
      );progn
   );if
   (princ)
);defun
;;;*-----------------------------------------------
(defun C:LOGOUT ( / a c d file fp)
   (setq a (TODAY)
      TIME2 (TIME)
      c (getvar "DWGNAME")
      d (strcat "Drg Exit    " a "  -  " TIME2 "  -  " c)
   );setq
   (if (/= c "Drawing.dwg")
      (progn
         (setq file (findfile "LOG.TXT")
            fp (open file "a")
         );setq
         (princ d fp)
         (princ "\n" fp)
         (close fp)
         (princ (strcat "\nLogged out at : " TIME2))
         (etime)
      );progn
   );if
   (princ)
);defun
;;;*-----------------------------------------------
(defun ETIME ( / hr1 m1 s1 tot1 hr2 m2 s2 tot2 total ht mt file fp)
   (setq hr1 (* 60 (* 60 (atof (substr time1 1 2))))
      m1 (* 60 (atof (substr time1 4 2)))
      s1 (atof (substr time1 7 2))
      tot1 (+ hr1 m1 s1)
      hr2 (* 3600 (atof (substr time2 1 2)))
      m2 (* 60 (atof (substr time2 4 2)))
      s2 (atof (substr time2 7 2))
      tot2 (+ hr2 m2 s2)
      total (- tot2 tot1)
```

```
      hr1 (/ total 3600)
      ht (fix hr1)
      hr1 (- hr1 ht)
      mt (* hr1 60)
      ht (rtos ht)
      mt (rtos mt)
   );setq
   (setq d (strcat "Editing Time This Session :
  " ht " Hours and " mt " minutes"))
   (setq file (findfile "LOG.TXT")
      fp (open file "a")
   );setq
   (princ d fp)
   (princ "\n" fp)
   (princ "==========================
==================================" fp)
   (princ "\n" fp)
   (close fp)
   (princ)
);defun
;;;*----------------------------------------
;;;*Calculates the Current Date
(defun TODAY ( / d yr mo day)
     (setq d (rtos (getvar "CDATE") 2 6)
          yr (substr d 3 2)
          mo (substr d 5 2)
         day (substr d 7 2)
     );setq
     (strcat day "/" mo "/" yr)
);defun
;;;*----------------------------------------
;;;*Calculates the Current Time
(defun TIME ( / d hr m s)
     (setq d (rtos (getvar "CDATE") 2 6)
          hr (substr d 10 2)
           m (substr d 12 2)
           s (substr d 14 2)
     );setq
     (strcat hr ":" m ":" s)
);defun
(princ)
```

Load the file and type "Login" to run it. Leave it for a minute or so and then type "Logout" to exit the routine.
Have a look at the file Log.txt. It should look something like this :

```
Drg Start   07/12/98  -  15:36:31  -  F4443.dwg
```

```
Drg Exit    07/12/98  -  15:36:34  -  F4443.dwg
Editing Time This Session :  0 Hours and 0.05 minutes
============================================================
```

Every time you log on and off the Starting Time, Ending Time and Total Editing Time will be appended to this file.

If you wish you can load Login.Lsp from your AcadDoc.Lsp file, and edit the Acad.mnu to call the Logout.Lsp routine before Exiting, Quiting or Starting a new drawing.

For more information on this topic, refer to the External Data Tutorial.

External Data

**Do you have a library full of blocks taking up vast amounts of disk space?
Do you find it difficult to locate a specific block because you've got so many or cannot remember what you called it?
By using external data you can parametrically draw these objects. If you look at my Structural Steel Program DDSTEEL you will notice that every section is drawn using data retrieved from an external data file.
The following tutorial will show you how to retrieve external data, format it into something that Autolisp can use, then place the data into individual variables.**

**First of all you need to create the external data file.
Create a text file called Test.dat and type in the following data :**

```
*6
152.0,10.0,124.0
*7
178.0,12.0,135.0
*8
203.0,14.0,146.0
*9
229.0,16.0,158.0
```

**This type of data file is known as a comma deliminating text file.
The data represents values that we would use to draw a specific object.
The first line (*6) represents the name of the object
and the second line (152.0,10.0,124.0) are the values that we are attempting to retrieve.**

Next, create a Lisp file called Test.Lsp and type in the following :

```
(defun C:Test (/ item data dline maxs
                 count chrct numb size)
  (setq size
    (getstring "\nEnter Size <6, 7, 8 or 9>: "))
;enter size req'd
  (setq dlist nil
        size  (strcat "*" size)
;add asterix to size
        file  (findfile "test.dat")
;find data file
        fp    (open file "r")
;open file to read
        item  (read-line fp)
;first line is label for file
  );setq
  (while item
;process each line of file
```

```
    (if (= item size)
;compare values
      (setq data (read-line fp)
;read a line
            item nil
;stop searching for item
      );setq
      (setq item (read-line fp))
;keep searching for item
    );if
  );while
  (if data
;if the size has been found
    (progn
      (setq maxs  (strlen data)
;establish length of input
            count 1
;initiliaze count
            chrct 1
;initiliaze char position
      );setq
      (while (< count maxs)
;process string one chr at a time
        (if (/= "," (substr data count 1))
;look for the commas
          (setq chrct (1+ chrct))
;increment to next position
          (setq numb  (atof
               (substr data
                  (1+ (- count chrct)) chrct))
;convert to real
                dlist (append dlist (list numb))
;add it to the list
                chrct 1
;resets field ct
          );setq
        );if
        (setq count (1+ count))
;increment the counter
      );while
      (setq numb  (atof
           (substr data
                  (1+ (- count chrct))))
;convert to real
            dlist (append dlist (list numb))
;add it to the list
      );setq
    );progn
```

```
  );if data
  (close fp)
;close data file
(mapcar 'set '(a b c) dlist)
;allocate to variables
);defun
```

The program basically does the following :

- Gets the name of the object who's data we want to retrieve. (eg 6)
- Adds a * to the front of it. (eg *6)
- Searches for the Data file (Test.Dat) and opens it to read.
- Reads each line of the file until it finds one matching our objects name.
- Reads the next line and stores the data.

Unfortunately, the data we have looks something like this :

(152.0,10.0,124.0)

Oh No, Commas...Don't worry the next section of the program deals with them. It parses the data and removes all commas so that we end up with a LIST looking like this :

(152.0 10.0 124.0)

Now by using the mapcar function we can allocate each item in the list to its own variable.

(mapcar 'set '(a b c) dlist)

I have deliberately not declared these variables as locals so that you can view them within AutoCAD.

Type *!dlist* to view the data list.
Type *!a* to see the first item in the list.
Type *!b* to see the second item in the list.
Type *!c* to see the third item in the list.

You can now use this data to draw your object.

As you can see, this is a much more efficient way of drawing objects that are the same shape but have differing dimensions.

List Manipulation

As you are probably well aware, LISP stands for "List Processing".
(Not "Lost in Stupid Parenthesis")
A list is a group of elements consisting of any data type and is stored as a single variable. A list can contain any number of Reals, Integers, Strings, Variables and even other Lists.
Let's have a look at a list. Type this :

```
(setq pt1 (getpoint "\nChoose a Point : "))
```

AutoLisp should return something like this :

```
(127.34 35.23 0.0)
```

Fine, you say, I've got a list but what do I do with it?
AutoLisp has many functions available to manipulate lists.
Let's have a look at them.

Car

The primary command for taking a list apart is the "Car" function.
This function returns the first element of a list. (The x coordinate.)
For example :

```
(setq a (car pt1))
```

Would return :

```
(127.34)
```

Cdr

This function returns the second element plus the remaining elements of a list. For example :

```
(setq b (cdr pt1))
```

Would return :

```
(35.23 0.0)
```

But what if we only wanted the second element? We could write :

```
(setq b (car (cdr pt1)))
```

But there is a better way. AutoLisp has provided the "Cadr" function which is basically an abbreviation of a nested command.

Cadr

This returns the second element of a list. (The y coordinate.)

```
(setq b (cadr pt1))
```

This would return :

```
(35.23)
```

Likewise, there is another abbreviated function to return the third element.

Caddr

This returns the third element of a list. (The z coordinate.)

```
(setq c (caddr pt1))
```

Would return :

```
(0.0)
```

AutoLisp has other functions that will retrieve values from lists of more than three elements. (Caar, cadar, etc). You can, though, us another function to access any element of a list. This is the "nth" function.

nth

The syntax for the nth function is as follows :

```
(nth num list)
```

"num" is the number of the element to return. Just remember that zero is the first element. For example given the list :

```
(setq d '("M10" "M20" "M30" 10.25))
(setq e (nth 0 d))
```

Would return :

```
("M10")
```

And likewise :

```
(setq f (nth 3 d))
```

Would return :

```
(10.25)
```

Next we will look at a practical example of using these functions.

We've now managed to extract elements from a list, but what do you do if you want to create a new list. Let's say you have two elements :

```
(setq a 200.0)
(setq b 400.0)
```

You want to combine them to create a new list. To do this you would use the "List" function. For example :

```
(setq c (list a b))
```

Would return :

```
(200.0 400.0)
```

You could also write the function like this :

```
(setq c '(a b))
```

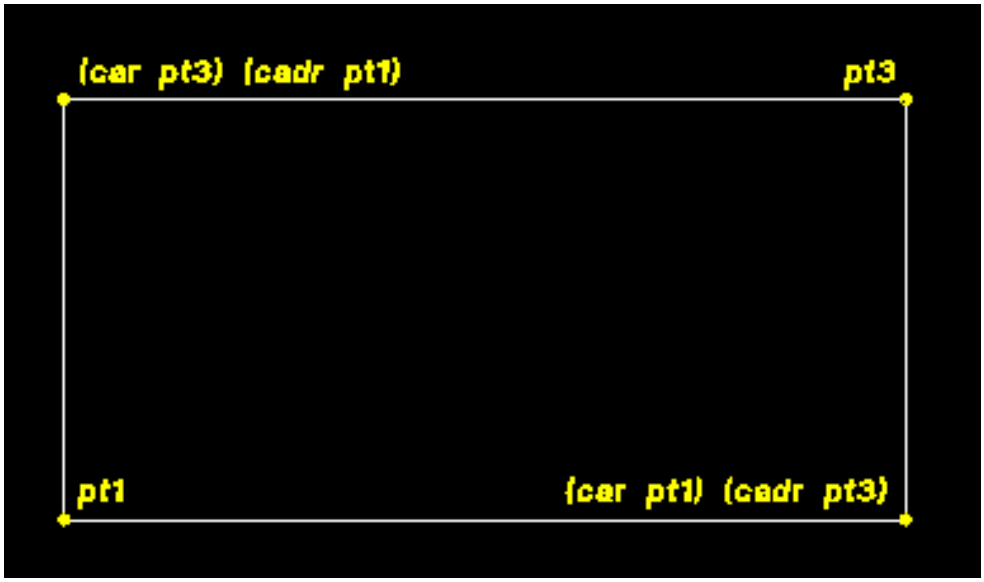
Here's an example of List Manipulation. We are going to use the (car), (cadr) and (list) function to draw a rectangle.

```
(defun c:rec ( / pt1 pt2 pt3 pt4)

        (setq pt1 (getpoint "\nSelect First Corner: "))
        ;get the first point

        (setq pt3 (getcorner pt1 "\nSelect Second Corner: "))
        ;get the third point

        (setq pt2 (list (car pt1) (cadr pt3)))
        ;construct the second point

        (setq pt4 (list (car pt3) (cadr pt1)))
        ;construct the fourth point

        (command "Line" pt1 pt2 pt3 pt4 "c")
        ;draw the rectangle

  (princ)

);defun
;*************************************************************
```

Let's look closer at the line :

```
(setq pt2 (list (car pt1) (cadr pt3)))
```

This function retrieves the first element (x coordinate) from the list pt1, the second element (y coordinate) from the list pt3, creates a list from these elements and stores the list in the variable pt2.
The following diagram should help you to better understand this.



AutoLisp provides other functions to manipulate lists. Let's have a look at some of them.

Append

This takes any number of lists and runs them together as one list :

```
        (append '(12.0 15.5) '("M20" "M30"))
```

Would return :

```
        (12.0 15.5 "M20" "M30")
```

Last

Will return the last element of a list :

```
        (last '("M20" "M24" "M30"))
```

Would return :

```
        ("M30")
```

Length

This returns an integer indicating the number of elements in a list :

```
(length '("M20" "M24" "M30"))
```

Should return :

```
(3)
```

Member

This function searches a list for a specific element. If found it returns the element plus the remainder of the list :

```
(member 'c '(a b c d e f))
```

would return :

```
(c d e f)
```

Reverse

Returns a list with it's elements reversed :

```
(reverse '(a b c d e f))
```

Will Return :

```
(f e d c b a)
```

Subst

Searches a list for an old element and returns a copy of the list with the new item substituted in place of every occurrence of the old item :

```
Syntax : (subst newitem olditem lst)

(setq lst '(a b c d e c)))

(subst 'f 'c lst)
```

Would return

```
(a b f d e f)
```

In the next section, we will have a look at a more advanced List Manipulation Example.

It is good practice (and manners) when writing Lisp routines to restore the system environment to the state that your program found it in on completion of your application. Most AutoLisp routines start and end like this :

```
(defun c:example ()
        (setq oldhigh (getvar "Highlight")
              oldsnap (getvar "Osmode")
              oldblip (getvar "BlipMode")
              oldecho (getvar "Cmdecho")
        );setq

        (setvar "Highlight" 0)
        (setvar "Osmode" 517)
        (setvar "Blipmode" 0)
        (setvar "Cmdecho" 0)

        Programme statements.............
        ................................

        (setvar "Highlight" oldhigh)
        (setvar "Osmode" oldsnap)
        (setvar "Blipmode "oldblip)
        (setvar "Cmdecho" oldecho)
  (princ)
);defun
;*************************************************
```

I must have written statements like this a thousand times in my Lisp routines. The following example is designed to act as a global routine that first stores, then changes specific system variables. On completion of the routine, the function is then called again and all system variables are returned to their previous settings.

```
(defun varget ()

        (setq lis '("HIGHLIGHT" "BLIPMODE" "CMDECHO"
                    "BLIPMODE" "OSMODE"))
        ;store names of system variables

        (setq  var (mapcar 'getvar lis))
        ;get the value of the system variables and
        ;store them as a list

        (setq var1 '(0 0 0 0 517))
        ;store the new values of the system variables

        (setq no 0)
        ;set counter to zero

        (repeat (length lis)
```

```
        ;get the number of variables in the list
        ;to use as the counter control number

                (setvar (nth no lis) (nth no var1))
                ;set the variables to their new values

                (setq no (1+ no))
                ;move up one in the list

        );repeat

  (princ);finish quietly

);defun

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

(defun varset ()

        (setq no 0)
        ;set counter to zero

        (repeat (length lis)
        ;get the number of variables in the list

                (setvar (nth no lis) (nth no var))
                ;reset the variables to their original values

                (setq no (1+ no))
                ;move up one in the list

        );repeat

  (princ);finish quietly

);defun

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

(princ);load quietly
```

Our Autolisp routine could now look like this :

```
(defun c:example ()

        (varget)
        ;store system variables and then reset them

        Programme statements.............
        ................................
```

```
        (varset)
        ;restore system variables

  (princ)
);defun
;*************************************************
```

As you can see, we have reduced the size of our routine by a lot and saved ourselves quite a bit of typing. These two routines could both be loaded from our AcadDoc.Lsp file so that they would be available to all of your routines.

Into the Database

Hold onto your hat because we're going to dive straight in here.
Fire up AutoCad and draw a line anywhere on the screen.
Now type this then press Enter:

```
(setq a (entlast))
```

Lisp should return something like this:

```
&ltEntity name: 2680880>
```

This is the Entity Name of the Line that you have just drawn.
Now type "Erase" then !a and press Enter twice.
The line should disappear. Type "OOPS" to bring it back.
You have just modified the AutoCAD Database.

Now type this :

```
(setq b (entget a))
```

This will retrieve the Entity Data. It should look something like this :

```
((-1 . &ltEntity name: 2680880>) (0 . "LINE") (5 . "270")
(100 . "AcDbEntity") (67 . 0) (8 . "0") (100 . "AcDbLine")
(10 400.378 621.82 0.0) (11 740.737 439.601 0.0)
(210 0.0 0.0 1.0))
```

Now type this line:

```
(setq c (cdr (assoc 10 b)))
```

Lisp should return:

```
(400.378 621.82 0.0)
```

Before you type the next few lines, make sure that your snap is turned off.

```
Command: circle
3P/2P/TTR/&ltCenter point>: !c
Diameter/&ltRadius> &lt10.0>: 20
```

A circle should be drawn at the end of the line.

This exercise was just to show you the ability of AutoLisp to go behind your drawing, into the database, and manage graphic and non-graphic information. This tutorial will show you how entities are stored and referenced in the database.
It will show you how AutoLisp reveals data about entities and how they can be modified and manipulated. Say au revoir to your wife and kid's and let's visit the AutoCad Database. (Star Trek theme music now starts.....)

When you first start to delve into the AutoCAD database it is, I admit, quite daunting. But, although entity access and manipulation is fairly complex, it can be divided into component parts that make it much easier to understand.
Let's have a look at an AutoLisp routine that can be used, as a sort of template which you can apply to numerous, similar applications. Have a close look at this coding :

```
(defun C:CHLAYER ( / a1 a2 n index b1 b2 b3 d1 d2)

        (prompt "\nSelect Entities to be Changed : ")
        (setq a1 (ssget))
        (prompt "\nPoint to Entity on Target Layer : ")
        (setq a2 (entsel))
        (setq n (sslength a1))
        (setq index 0)
        (setq b2 (entget (car a2)))
        (setq d2 (assoc 8 b2))
        (repeat n
                (setq b1 (entget (ssname a1 index)))
                (setq d1 (assoc 8 b1))
                (setq b3 (subst d2 d1 b1))
                (entmod b3)
                (setq index (1+ index))
        );repeat
   (princ)
);defun
(princ)
```

This routine allows you to select any number of objects and change them to a different layer. The target layer is chosen by simply pointing to an object on the desired layer. (To test this routine, you will need to create a drawing with objects on different layers.) Let's have a look line by line :

```
(defun C:CHLAYER ( / a1 a2 n index b1 b2 b3 d1 d2)
```

Defines the function and declares all variables as local.

```
        (prompt "\nSelect Entities to be Changed : ")
```

Prompts the user.

```
        (setq a1 (ssget))
```

Allows the user to select the objects to be changed. The selection set is assigned to variable 'a1'.

```
        (prompt "\nPoint to Entity on Target Layer : ")
```

Prompts the user to select the Target Layer.

```
(setq a2 (entsel))
```

This is a special type of selection statement that only allows you to select one entity.

```
(setq n (sslength a1))
```

Counts the number of entities in the selection set 'a1' and stores this number in variable 'n'.

```
(setq index 0)
```

Sets the loop control variable 'index' to zero.

```
(setq b2 (entget (car a2)))
```

This statement retrieves the entity list from 'a2' and assigns it to 'b2'.

```
(setq d2 (assoc 8 b2))
```

This looks for the code 8 in the entity list 'a2', and then assigns the sub list to 'd2'.

```
(repeat n
```

This begins the loop that pages through the selection set.

```
        (setq b1 (entget (ssname a1 index)))
```

This gets the entity list and assigns it to 'b1'.

```
        (setq d1 (assoc 8 b1))
```

Gets the sublist code 8. (The Layer)

```
        (setq b3 (subst d2 d1 b1))
```

Substitutes the new 'd2' layer for the old 'd1' layer in the entity list 'a1', and assigns it to the new entity list 'b3'.

```
        (entmod b3)
```

Updates the new entity list in the database.

```
        (setq index (1+ index))
```

Increases the 'index' variable by 1, priming it for the next loop.

```
        );repeat
```

Closes the repeat loop.

```
   (princ)
```

Finish cleanly.

```
);defun
```

Closes the function.

```
(princ)
```

Clean Loading.

Listed below is another routine that allows you to globally change the height of text without affecting other entities. As you will see, the only difference is, is that we have added a conditional filter to the routine.

```
(defun C:CHGTEXT ( / a ts n index b1 b2 b c d)

        (setq a (ssget))
        (setq ts (getreal "\nEnter New Text Height : "))
        (setq n (sslength a))
        (setq index 0)
        (repeat n
                (setq b1 (entget (ssname a index)))
                (setq index (index+ 1))
                (setq b (assoc 0 b1))
                (if (= "TEXT" (cdr b))
                        (progn
                                (setq c (assoc 40 b1))
                                (setq d (cons (car c) ts))
                                (setq b2 (subst d c b1))
                                (entmod b2)
                        );progn
                );if
        );repeat
   (princ)
);defun
(princ)
```

Well I bet your brain hurts after that lot!!!
Next we'll have a quick look at Tables.

The AutoCad Database does not only consist of entities but also includes several other sections, such as the Tables Section.
Tables store information about entities that are maintained globally within the drawing. For example, when you insert a block into a drawing, how does AutoCAD know what the block looks like? The definition of a block is stored in the Block Table. What happens, for example, if you need to create a layer? You have to know if the layer already exist because if you try to create a layer that already exists your program will crash. Therefore, you would search the Layers Table first to see if the layer exists.

There are nine (9) Tables that you can access :

```
Layer Table                          "LAYER"
Linetype Table                       "LTYPE"
Named View Table                     "VIEW"
Text Style Table                     "STYLE"
Block Table                          "BLOCK"
Named UCS Table                      "UCS"
Named Application ID Table           "APPID"
Named Dimension Style Table          "DIMSTYLE"
Vport Configuration Table            "VPORT"
```

A Table is split into 2 parts: The 'names' of the entries in the Table and the 'details' of each entry. For example, in the Layers Table, the name of the entries would be the names of the layers that exist. The details of an individual layer would be colour, linetype, on, off, frozen, thawed, locked, unlocked or current.

To access a Table we would use the (tblsearch) function. Let's have a look at an example :

Assume that you want to know whether a layer called STEEL exists in your drawing. First create the layer STEEL then type the following :

```
(setq t (tblsearch "LAYER" "STEEL"))
```

The entity list of the layer STEEL should be returned :

```
((0 . "LAYER") (2 . "STEEL") (70 . 64) (62 . 7) (6 . "CONTINOUS"))
```

The first part of the entity list is '0', indicating Associative 0.
In this case it's an entry in the "LAYER" Table.
Associative 2 indicates the name of the layer. STEEL in our case.
Associative 70 is the state of the entity. 1 is Frozen, 2 is Frozen on new paper space view ports and 4 is locked. These numbers are added to 64. In this case the layer is neither frozen nor locked.
Associative 62 is the colour of the layer. Ours is white which is colour number 7. If the colour number is a negative number then the layer is off.
Associative 6 is the linetype of the layer, in this case, "CONTINUOUS".

If the (tblsearch) had not found the layer then it would have returned 'nil' and you would know that the layer did not exist.

Sometimes you don't know the name of a layer or a block but you need a list of them. This is when the (tblnext) function comes into play.

Let's assume that 4 layers exist in your drawing. The layer names are MECH, STEEL, PIPE and TXT. Enter the following :

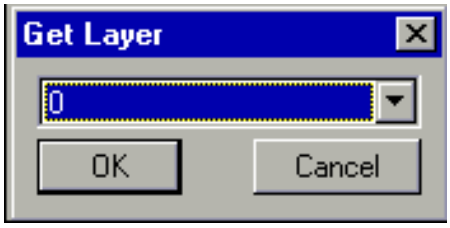
```
(tblnext "Layer")
```

AutoLISP should return something like this :

```
((0 . "LAYER") (2 . "PIPE") (70 . 0) (62 . 7) (6 . "CONTINUOUS"))
```

Repeat the same command four additional times. You will get a new entity list for each of the layers.
The last time you type the command AutoLISP will return 'nil' because there are no more additional layers.

Let's have a look at this in action. We are going to design a dialogue box that displays a drop down list of all the available layers in a specific drawing. The user can then choose any layer which can then be used within his routine.



The Dialogue Coding looks like this :

```
getlay  : dialog {                          //dialog name
        label = "Get Layer" ;               //give it a label

        : popup_list {                      //define list box
        key = "sel1";                       //give it a name
        value = "0";                        //initial value
        }                                   //end list
```

```
        ok_cancel ;                                  //predifined OK/Cancel

}                                                    //end dialog
```

Now the AutoLisp Coding :

```
(defun c:getlay ( / NAMES SIZ1)
;define funcion

        (setvar "cmdecho" 0)
        ;switch off command echo

        (setq siz1 "0")
        ;initiliase variable

        (setq userclick T)
        ;set flag

        (setq f 1)
        ;rewind pointer

        (while
        ;start while loop

          (setq t1 (tblnext "Layer" f))
          ;get the next layer

          (setq f nil)
          ;reset pointer

          (setq b (cdr (assoc 2 t1)))
          ;get the layer name

            (if (/= b "DEFPOINTS")
            ;if name is not equal to DEFPOINTS

               (setq NAMES (append NAMES (list b)))
               ;Add the layer name to the list

            );if

        );while


        (setq dcl_id (load_dialog "getlay.dcl"))
        ;load dialogue

        (if (not (new_dialog "getlay" dcl_id)
```

```
      ;check for errors

            );not

           (exit)
           ;if problem exit

      );if

(set_tile "sel1" "0")
;initilise list box

(start_list "sel1")
;start the list

(mapcar 'add_list NAMES)
;add the layer names

(end_list)
;end the list

  (action_tile

  "cancel"

  "(done_dialog) (setq userclick nil)"

  );action_tile
  ;if cancel set flag to nil

(action_tile

  "accept"

  (strcat

    "(progn

      (setq SIZ1 (get_tile \"sel1\"))"

     " (done_dialog)(setq userclick T))"

  );strcat

);action tile
;if OK get the layer selected

(start_dialog)

(unload_dialog dcl_id)

 (if userclick
```

```
   ;if flag true

    (progn

     (setq SIZ1 (nth (atoi SIZ1) NAMES))
     ;get the name of the layer from the list

     (alert (strcat "\nYou Selected Layer  " SIZ1))
     ;display the name

    );end progn

   );end if

(princ)

);defun C:getlay

(princ)
```

DXF Group Codes

The following table gives the group code or group code range accompanied by an explanation of the group code value. In the table, "fixed" indicates that this group code always has the same purpose. The purpose of group codes that aren't fixed can vary depending on the context.

Entity group codes by number :

Group code	Description
-5	APP: persistent reactor chain
-4	APP: conditional operator (used only with ssget)
-3	APP: extended data (XDATA) sentinel (fixed)
-2	APP: entity name reference (fixed)
-1	APP: entity name. This changes each time a drawing is opened. It is never saved. (fixed)
0	Text string indicating the entity type (fixed)
1	Primary text value for an entity
2	Name (attribute tag, block name, and so on)
3-4	Other textual or name values
5	Entity handle. Text string of up to 16 hexadecimal digits (fixed)
6	Linetype name (fixed)
7	Text style name (fixed)
8	Layer name (fixed)
9	DXF: variable name identifier (used only in HEADER section of the DXF file).
10	Primary point. This is the start point of a line or text entity, center of a circle, and so on. DXF: X value of the primary point (followed by Y and Z value codes 20 and 30) APP: 3D point (list of three reals)
11-18	Other points. DXF: X value of other points (followed by Y value codes 21-28 and Z value codes 31-38) APP: 3D point (list of three reals)
20, 30	DXF: Y and Z values of the primary point
21-28, 31-37	DXF: Y and Z values of other points
38	DXF: entity's elevation if nonzero. Exists only in output from versions prior to AutoCAD Release 11.
39	Entity's thickness if nonzero (fixed)
40-48	Floating-point values (text height, scale factors, and so on)
48	Linetype scale. Floating-point scalar value. Default value is defined for all entity types.
49	Repeated floating-point value. Multiple 49 groups may appear in one entity for variable-length tables (such as the dash lengths in the LTYPE table). A 7x group always appears before the first 49 group to specify the table length.
50-58	Angles (output in degrees to DXF files and radians through AutoLISP and ARX applications).
60	Entity visibility. Integer value. Absence or 0 indicates visibility; 1 indicates invisibility.
62	Color number (fixed)
66	"Entities follow" flag (fixed)
67	Space--that is, model or paper space (fixed)
68	APP: identifies whether viewport is on but fully off screen; is not active or is off.
69	APP: viewport identification number.
70-78	Integer values, such as repeat counts, flag bits, or modes
90-99	32-bit integer values
100	Subclass data marker (with derived class name as a string). Required for all objects and entity classes that are derived from another concrete class to segregate data defined by different classes in the inheritance chain for

	the same object. This is in addition to the requirement for DXF names for each distinct concrete class derived from ARX (see "Subclass Markers").
102	Control string, followed by "{" or "}". Similar to the xdata 1002 group code, except that when the string begins with "{", it can be followed by an arbitrary string whose interpretation is up to the application. The only other allowable control string is "}" as a group terminator. As noted before, AutoCAD does not interpret these strings except during drawing audit operations; they are for application use.
105	DIMVAR symbol table entry object handle
210	Extrusion direction (fixed). DXF: X value of extrusion direction APP: 3D extrusion direction vector
220, 230	DXF: Y and Z values of the extrusion direction
280-289	8-bit integer values
300-309	Arbitrary text strings
310-319	Arbitrary binary chunks with same representation and limits as 1004 group codes: hexadecimal strings of up to 254 characters represent data chunks of up to 127 bytes.
320-329	Arbitrary object handles. Handle values that are taken "as is." They are not translated during INSERT and XREF operations.
330-339	Soft-pointer handle. Arbitrary soft pointers to other objects within same DXF file or drawing. Translated during INSERT and XREF operations.
340-349	Hard-pointer handle. Arbitrary hard pointers to other objects within same DXF file or drawing. Translated during INSERT and XREF operations.
350-359	Soft-owner handle. Arbitrary soft ownership links to other objects within same DXF file or drawing. Translated during INSERT and XREF operations.
360-369	Hard-owner handle. Arbitrary hard ownership links to other objects within same DXF file or drawing. Translated during INSERT and XREF operations.
999	DXF: The 999 group code indicates that the line following it is a comment string. DXFOUT does not include such groups in a DXF output file, but DXFIN honors them and ignores the comments. You can use the 999 group to include comments in a DXF file that you've edited.
1000	ASCII string (up to 255 bytes long) in extended data.
1001	Registered application name (ASCII string up to 31 bytes long) for extended data.
1002	Extended data control string ("{"or "}").
1003	Extended data layer name.
1004	Chunk of bytes (up to 127 bytes long) in extended data.
1005	Entity handle in extended data. Text string of up to 16 hexadecimal digits
1010	A point in extended data DXF: X value (followed by 1020 and 1030 groups) APP: 3D point
1020, 1030	DXF: Y and Z values of a point
1011	A 3D world space position in extended data DXF: X value (followed by 1021 and 1031 groups) APP: 3D point
1021, 1031	DXF: Y and Z values of a World space position
1012	A 3D world space displacement in extended data DXF: X value (followed by 1022 and 1032 groups) APP: 3D vector
1022, 1032	DXF: Y and Z values of a World space displacement
1013	A 3D world space direction in extended data. DXF: X value (followed by 1022 and 1032 groups) APP: 3D vector
1023, 1033	DXF: Y and Z values of a World space direction
1040	Extended data floating-point value.
1041	Extended data distance value.
1042	Extended data scale factor.
1070	Extended data 16-bit signed integer.
1071	Extended data 32-bit signed long.

Selection Sets.

When you work with AutoCad, you very seldom work with only one entity or, for that matter, one type of entity. To be able to work efficiently with a group, or selection set, of entities, you need to be able to place them in one place and work on them as a group. You might also want to filter the entities so that only a certain type is within the group.
The AutoLisp function that enables you to do this is the (ssget) function.

```
(setq sel1 (ssget))
```

This function allows you to select as many entities as you like using any selection method, such as Window, Crossing, Fence, etc.
You can also include a selection set filter within the function.

```
(setq sel1 (ssget "x"))
```

The "x" following the (ssget) is the most basic of filters. This filter selects ALL. This is quite sufficient in some circumstances but you can filter the entities further by following the "x" with a list of associative codes. For example :

```
(setq sel1 (ssget "x" '((8 . "STEEL"))))
```

This expression would create a selection set containing only entities that are on the STEEL layer. (Associative code 8 refers to the Layer.)
You can use more than one associative code if you desire :

```
(setq sel1 (ssget "x" '((0 . "CIRCLE")(8 . "STEEL"))))
```

The above example would create a selection set of all CIRCLES on the Layer STEEL.

If you don't want to search the entire drawing for a selection set, but would rather use a select by Window or Crossing, then just omitt the "x" option.

```
(setq sel1 (ssget '((0 . "CIRCLE")(8 . "STEEL"))))
```

This example permits the user to select entities using any selection method he prefers but, will only create a selection set of CIRCLES on Layer STEEL.

Try this next :

```
(setq sel1 (ssget "w" "\nSelect Objects by Window: "))
```

Doesn't work, does it?
You cannot use prompts within the (ssget) function. But, you can use other AutoCad functions to feed the required information to the (ssget) function.

```
(prompt "\nSelect Objects by Window")
(setq p1 (getpoint "\nFirst Corner: "))
(setq p2 (getpoint p1 "\nSecond Corner: "))
```

```
(setq sel1 (ssget "w" p1 p2))
```

You can also use other selection options with (ssget)

```
(setq sel1 (ssget "P"))
```

**This will create a selection set of all "Previous" entities.
Other options available are "L" for "Last" and "I" for "Implied".**

**You can also use logical filters when creating selection sets.
Have a look at a previous example :**

```
(setq sel1 (ssget '((0 . "CIRCLE")(8 . "STEEL"))))
```

What we are saying here is :

**"Create a selection set of all the CIRCLES from the entities selected
AND they must be on Layer "STEEL".**

AND is the default logical filter when you string associative codes together.

**You can also use the OR logical filter, but to do this you must inform AutoLisp
first. To do this you use a special type of associative code, the -4.**

```
(setq sel1 (ssget '((-4 . "OR")(8 . "STEEL")(8 . "PIPE")(-4 . "OR>"))))
```

This would creat a selection set of all entities on Layer STEEL "OR" on Layer PIPE.

As well as logical filters, you can also have relation filters :

```
(setq sel1 (ssget '(0 . "CIRCLE")(-4 . ">=")(40 . 2.0)))
```

**This would creat a selection set of all CIRCLES with a RADIUS (group 40) of
greater or equal to 2.0.**

All of these different types of filters can be nested.

**You can also Add and Delete entities from selection sets. Three guesses what
these function names are? You were right first time, (ssadd) and (ssdel).**

```
(setq sel1 (ssget))
;create the first selection set

(setq ent (car (entsel))
;select an entity and use (car (ensel))
; to retrieve the entity name
```

```
        (setq sel1 (ssadd ent sel1))
        ;add the entity to the selection set
```

To delete an entity is exactly the same except for the last line :

```
        (setq sel1 (ssdel ent sel1))
```

But what would you do if you wanted to add two selection sets together? The following explains how to create a "Union" between 2 selection sets :

```
        (setq ct 0)
        ;set counter to zero

        (repeat (sslength sel2)
        ;get the number of items in selection set 2
        ;and loop that number of times

        (ssadd (ssname sel2 ct) sel1)
        ;get the name of the entity from selection set 2
        ;by using the counter index number and add it to
        ;selection set 1

        (setq ct (1+ ct))
        ;increment the counter by 1

        );end repeat
```

Here's an example of a simple routine that will count the number of blocks contained within a drawing :

```
(defun c:bcount ( / p1 b a n)

        (setq p1 (getstring "\Name of Block : "))
        ;get the name of the block

        (setq b (cons 2 p1))
        ;construct a dotted pair - code 2 is for blocks

        (setq a (ssget "x" (list b)))
        ;filter for the block name

        (if (/= a nil)
        ;check if there are any blocks of that name

           (progn
           ;if there is...

                (setq n (sslength a))
                ;count the number of blocks

                (alert (strcat "\nThere are " (itoa n) " in the DataBase"))
```

```
                ;display the result

            );progn
            ;if there are no blocks

                (alert "\nThere are none in the DataBase")
                ;inform the user

        );if

    (princ)

);defun

(princ)
```

Well that's about it with selection sets. Just remember that you can save yourself an awful lot of work by using selection sets with filters.
Filter at the source rather than programmatically trying to filter out the undesirable entities at a later stage. Ta Ta for now.....

Selection Set Filters

*Written by **Jarvis Fosdick***

Very often I will want to select all items on a layer and move them off to the right hand side of my workspace, but what happens if I want to select two layers? There are always several ways to do things. I have found using the **filter** command is somewhat cumbersome. Another way to use these filters is from the **ssget** function.

Command: copy

Select objects: ***(ssget '((8 . "mylayer") (0 . "circle")))***

This will only allow you to select circles on mylayer. If you wanted to select all the circles on layers mylayer and mylayer2 it would look as follows.

Command: copy

Select objects: ***(ssget '((0 . circle) (-4 . "<or") (8 . "mylayer") (8 . "mylayer2") (-4 . "or>")))***

The –4 dxf group code ***(-4 . "<or")*** begins the conditional operator. The less than or open alligator symbol < tells AutoCad to evaluate the conditional until it finds a closing alligator **>**. We use ***(-4 . "or>")*** to end the conditional.

Conditionals can be used together.

```
(ssget '(
(-4 . "<or")
(8 . "notes") (0 . "circle")
 (-4 . "<and")
        (8 . "s-boundary")(0 . "line")
         (-4 . "and>")
(-4 . "or>")
))
```

This would select only those entities that are on the layer notes or are circles and entities that are both lines on the layer s-boundary.

Conditional operators -- **AND**, **OR**, **XOR**, and **NOT** -- must be paired and balanced correctly in the filter list. The number of operands you can enclose depends on the operation. Here is a list of the Conditionals or "selection set filters" you can use:

"<AND" (One or more operands) **"AND>"**

(ssget '((-4 . "<and") (0 . "line") (8 . "text") (62 . 3) (-4 . "and>")))

Selects entities that match all conditions.

"<OR" (One or more operands) **"OR>"**

(ssget '((-4 . "<or") (0 . "line") (8 . "text") (62 . 3) (-4 . "or>")))

Selects entities that match any of the conditions.

"<XOR" (Two operands) **"XOR>"**

(ssget '((-4 . "<xor") (8 . "text") (62 . 3) (-4 . "xor>")))

Selects entities that match one or the other condition.

"<NOT" (One operand) **"NOT>"**

(ssget '((-4 . "<not") (0 . "line") (-4 . "not>")))

Selects entities that do not match one condition.

The **XOR** conditional works as an exclusive **OR** operator. For instance, using **OR** we may select entities that are text and are either on the layer notes or have the color 3 or both.

(ssget '((0 . "text") (-4 . "<or") (8. "notes) (62 . 3) (-4 . "or>")))

However, using **XOR** we may select only entities that are text and on layer notes or entities that are text and the color 3. We cannot select entities that are text on the layer notes and the color 3.

(ssget '((0 . "text") (-4 . "<xor") (8. "notes) (62 . 3) (-4 . "xor>")))

If we wish to select text that is both on the layer notes and the color 3 using **XOR** we must group these properties with another conditional.

```
(ssget '(
(0 . "text")
 (-4 . "<xor")
  (-4 . "<and") (8 . "notes") (62 . 3) (-4 . "and>")
 (-4 . "xor>")
))
```

We can nest all sorts of conditionals into a selection set filter:

```
(ssget '(
  (-4 . "<xor") (8 . "mylayer")
                (-4 . "<or") (0 . "text")
                             (-4 . "<xor")  (8 . "notes")
                                            (-4 . "<and") (62 . 2) (0 . "line") (-4 . "and>")
                             (-4 . “xor>")
          (-4 . "or>")
  (-4 . "xor>")
))
```

This is probably not practical, but it will work. See if you can figure out what it would select. There is not really all that much to these conditionals and they are very handy. I use them mostly at the command prompt to copy several layers at once, which tends to be easier then using the filter dialog box. The following lisp routine will let you copy all the objects from two layers.

```
(defun c:layer_copy ( / laa la p1 p2 ss )
  (princ "\n Choose an object on desired layer: ")
    (setq laa (assoc 8 (entget (car (entsel)))))
  (princ "\n Choose another object a different desired layer: ")
    (setq la (assoc 8 (entget (car (entsel)))))
 (setq ss (ssget "x" (list  (cons -4 "<or") laa la (cons -4 "or>") )))
 (command "copy" ss )
);defun
(princ)
```

Working with Layers & Styles.

I've had a lot of queries from people asking me how to create layers, change layers, change text styles, etc. using AutoLisp.
This tutorial will take you through the steps of doing just that.

Before you do anything in regards to layers and styles, it's always best to retrieve the current layer, or text style, so that you can restore it. To retrieve the current layer, use the following coding :

```
(setq oldlayer (getvar "CLAYER"))
```

This will retrieve the name of the current layer and store it in variable "oldlayer". To restore the previous layer is just as simple:

```
(setvar "CLAYER" oldlayer)
```

This will set the current layer to the previous layer name stored in variable "oldlayer".

You would use exactly the same syntax for retrieving and restoring the name of the current Text Style :

```
(setq oldstyle (getvar "TEXTSTYLE"))
(setvar "TEXTSTYLE" oldstyle)
```

You might think that this would be a good way of changing layers and style.
The problem here, is that if the layer name you want to change to does not exist your program will crash.

You can use the "tblsearch" function to test if a layer or style exists :

```
(tblsearch "LAYER" "TESTLAYER")
(tblsearch "STYLE" "MYSTYLE")
```

If Layer "TESTLAYER" or Style "MYSTYLE" exists in your drawing, tblsearch will return "True", otherwise it will return "Nil".

Here's an example that tests if a layer exist, and if it does, sets the current layer to that layer:

```
(setq oldlayer (getvar "clayer"))
;retrieve the current layer

(setq flag (tblsearch "LAYER" "NEWLAYER"))
;find out if the layer exists

(if flag
```

```
;if the layer exists

        (setvar "CLAYER" "NEWLAYER")
        ;set current layer to "NEWLAYER"

);end if
```

This though, is quite a lot of coding just to check if a layer exists.
A better way, that applies to both Layers and Text Styles is to use the command function:

```
(command "Layer" "M" "NEWLAYER" "")
(command "Style" "Italict" "Italict.shx" "" "" "" "" "" "")
```

Both of these examples will change the layer or style, but will also create a new layer or text style if the layer or style does not exist within the drawing. (The style .shx file must exist and be within the AutoCad search path.)

I've also been asked that when I want to place text, dimensions, etc. into my drawing, how do I ensure that they are drawn on a particular layer?
Now, you could modify the AutoCad Menu to achieve this, but I prefer to create a partial menu with the modified macros in place.

The way to go about this is to simply prefix the menu macro with one that changes to the specific layer :

```
[Dtext]^C^C^CLayer;M;5;;_Dtext
```

This ensures that I am on Layer 5 when I add text to a drawing.

Well, I hope this has helped you. Keep well.....

PolyLines and Blocks.

PolyLines and Blocks!!! Come back... Don't run away....
Honestly, they are a lot easier to deal with than you think.
I know that they are called "complex entities", but the only difference between them and other entities is that we just have to dig a bit deeper to get to what we want. In fact, once we get there I'll show you a couple of things that you swear is magic. So bear with me, take your time, and hang on for a ride on the magic carpet.....

PolyLines.

The lwpolyline entity, or "optimized polyline," is new to Release 14.
A lwpolyline is defined in the drawing database as a single graphic entity.
This is different than a standard polyline, which is defined as a group of subentities. Lwpolylines display faster and consume less disk space and RAM.
In Release 14, 3D polylines are always created as standard polyline entities.
2D polylines are created as lwpolyline entities unless they have been curved or fitted with the PEDIT command. When a drawing from a previous release is opened in Release 14, all 2D polylines convert to lwpolylines automatically unless they have been curved or fitted or contain xdata.

We will have a look at the R13 and below Polyline first.
First of all draw 3 joined polylines. (3DPoly if you are using R14).
Then type this :

```
Command: (setq e (entget (car (entsel))))
```

AutoLisp should return something like this :

```
Select object: ((-1 . ) (0 . "POLYLINE") (5 . "27B") (100
. "AcDbEntity") (67 . 0) (8 . "0") (100 . "AcDb3dPolyline") (66 . 1) (10 0.0
0.0 0.0) (70 . 8) (40 . 0.0) (41 . 0.0) (210 0.0 0.0 1.0) (71 . 0) (72 . 0) (73
. 0) (74 . 0) (75 . 0))
```

Hey, wait a minute!!!.....AutoLisp has returned the entity list, and I can see that it's a Polyline, but there are no co-ordinates, and where does AutoLisp get the co-ordinates for all the vertices?
As I said earlier, we need to dig a little bit deeper to get the information we require. This is where the (entnext) function comes into play.
Now type the following 5 code segments :

```
Command: (setq e1 (entget (entnext (cdr (car e)))))
((-1 . ) (0 . "VERTEX") (5 . "27C") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (100 . "AcDbVertex") (100 . "AcDb3dPolylineVertex") (10
391.774 521.633 0.0) (40 . 0.0) (41 . 0.0) (42 . 0.0) (70 . 32) (50 . 0.0) (71
. 0) (72 . 0) (73 . 0) (74 . 0))

Command: (setq e2 (entget (entnext (cdr (car e1)))))
((-1 . ) (0 . "VERTEX") (5 . "27D") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (100 . "AcDbVertex") (100 . "AcDb3dPolylineVertex") (10
758.971 383.418 0.0) (40 . 0.0) (41 . 0.0) (42 . 0.0) (70 . 32) (50 . 0.0) (71
. 0) (72 . 0) (73 . 0) (74 . 0))

Command: (setq e3 (entget (entnext (cdr (car e2)))))
```

```
((-1 . ) (0 . "VERTEX") (5 . "27E") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (100 . "AcDbVertex") (100 . "AcDb3dPolylineVertex") (10
257.549 377.344 0.0) (40 . 0.0) (41 . 0.0) (42 . 0.0) (70 . 32) (50 . 0.0) (71
. 0) (72 . 0) (73 . 0) (74 . 0))

Command: (setq e4 (entget (entnext (cdr (car e3)))))
((-1 . ) (0 . "VERTEX") (5 . "27F") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (100 . "AcDbVertex") (100 . "AcDb3dPolylineVertex") (10
391.774 521.633 0.0) (40 . 0.0) (41 . 0.0) (42 . 0.0) (70 . 32) (50 . 0.0) (71
. 0) (72 . 0) (73 . 0) (74 . 0))

Command: (setq e5 (entget (entnext (cdr (car e4)))))
((-1 . ) (0 . "SEQEND") (5 . "280") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (-2 . ))
```

The (cdr (car e)) returns the entity name of entity list e. Each code segment then uses the (entnext entity name) of the entity that precedes it.
Take note of the entity type of each variable :

```
e       (0 . "POLYLINE")
e1      (0 . "VERTEX")
e2      (0 . "VERTEX")
e3      (0 . "VERTEX")
e4      (0 . "VERTEX")
e5      (0 . "SEQEND")
```

Do you see that a 3 line Polyline consists of a master or, parent list, 4 vertex and an end-of sequence ("SEQEND") list.

To extract the entitity list for each vertex is therefore, quite easy. We just need to loop through the sequence of vertices until we reach the SEQEND list.
Here's an example of a function that will print the coordinates for each vertex of a Polyline :

```
(defun c:coord ( / e r)

        (setq e (entget (car (entsel))))
        ;get the parent entity list

        (setq r 1)
        ;set loop control number to 1

        (while r
        ;while loop control is not nil, carry on looping

           (setq e (entget (entnext (cdr (car e)))))
           ;get the vertex entity list

           (if (/= (cdr (assoc 0 e)) "SEQEND")
           ;if it is not "end-of-sequence

                (progn
                ;do the following

                  (terpri)
                  ;new line
```

```
                (princ (cdr (assoc 10 e)))
                ;print the co-ordinates

              );progn

              (setq r nil)
              ;if end of sequence, stop looping

           );if

        );while

  (princ)

);defun

(princ)
```

There is a quicker way of retrieving the entity list of a Polyline vertex.
(nentsel) let's you select an entity and returns the name of the entity even if it belongs to a polyline.
Try this. Type this, then select any vertex of a polyline.

```
(setq e (entget (car (nentsel))))
```

AutoLisp should return something like this :

```
Select object: ((-1 . ) (0 . "POLYLINE") (5 . "270") (100
. "AcDbEntity") (67 . 0) (8 . "0") (100 . "AcDb3dPolyline") (66 . 1) (10 0.0
0.0 0.0) (70 . 9) (40 . 0.0) (41 . 0.0) (210 0.0 0.0 1.0) (71 . 0) (72 . 0) (73
. 0) (74 . 0) (75 . 0))
```

How's that? Straight to the entity list!!

Now, while we're here, let's have a quick look at blocks.....

Blocks.

Create a block consisting of a couple of lines and a circle with a radius of 20.

Now type this :

```
(setq e (entget (car (nentsel))))
```

Now pick the circle. AutoLisp should return something like this:

```
Select object: ((-1 . ) (0 . "CIRCLE") (5 . "282") (100 .
"AcDbEntity") (67 . 0) (8 . "0") (100 . "AcDbCircle") (10 0.0 -1.51849 0.0) (40
. 20.0) (210 0.0 0.0 1.0))
```

Hey, Hang about. It's returned the entity list of the circle even though it's part of a block!! Now type the following:

```
(setq d (assoc 40 e))
(setq e1 (subst '(40 . 50.0) d e))
(entmod e1)
```

Now REGEN the drawing.
Did you noticed what happened? The radius of the circle has change even though it's part of a block. Not only that, it has also redefined the block definition.
In other words, every block in the drawing with the same name would have changed.
I told you it was magic.....
Let's change the layer of the circle:

```
(setq d (assoc 8 e1))
(setq e2 (subst '(8 . "STEEL") d e1))
(entmod e2)
```

Again, REGEN the drawing. The circle has changed to Layer "STEEL".
Now do you see what I mean by magic. I bet you never thought you would be able to modify a block without exploding it. Well you can now.....

LwPolylines.

As we mentioned earlier, LwPolylines are a new feature in AutoCad Release 14.
They differ in that they are defined as a single entity. Let's have a look at a LwPolylines entity list.
Draw a LwPolyline and enter this:

```
(setq e (entget (car (entsel))))
```

AutoLisp should return something like this:

```
Select object: ((-1 . ) (0 . "LWPOLYLINE") (5 . "287")
(100 . "AcDbEntity") (67 . 0) (8 . "0") (100 . "AcDbPolyline") (90 . 3) (70 .
1) (43 . 0.0) (38 . 0.0) (39 . 0.0) (10 597.908 661.3) (40 . 0.0) (41 . 0.0)
(42 . 0.0) (10 1179.86 476.045) (40 . 0.0) (41 . 0.0) (42 . 0.0) (10 479.39
397.084) (40 . 0.0) (41 . 0.0) (42 . 0.0) (210 0.0 0.0 1.0))
```

As you can see, there is no need to use (entnext) to step through the vertex entities as the group 10 entity code is already part of the parent list.
But, we do have another problem!! There are numerous group 10 entity codes.
(As well as Group 40 - Start Width; Group 41 - End Width and Group 42 - Bulge)
To extract these, we need to first, find the length of the list. Then we must loop through each code entity in the list searching for a group 10. Once found, we can easily extract the vertex co-ordinates. The following function does just that:

```
(defun c:lwcoord (/ e len n e1)

        (setq e (entget (car (entsel))))
        ;get the entity list

        (setq len (length e))
        ;get the length of the list

        (setq n 0)
```

```
        ;set counter to zero

        (repeat len
        ;repeat for the length of the entity list

          (setq e1 (car (nth n e)))
          ;get each item in the entity list
          ;and strip the entity code number

          (if (= e1 10)
          ;check for code 10 (vertex)

            (progn
            ;if it's group 10 do the following

                (terpri)
                  ;new line

                  (princ (cdr (nth n e)))
                  ;print the co-ordinates

            );progn

          );if
          (setq n (1+ n))
          ;increment the counter

        );repeat

  (princ)
);defun
(princ)
```

Well, that's it concerning Polylines and Blocks. I told you it was easy!!!
Remember, I've only scratched the surface with the things that you can do once you dig into entity lists, especially with complex entities.
One last thing. Want to create your own entity?
Normally, in AutoLisp you would draw a line like this:

```
        (command "Line" pt1 pt2 "")
```

Now, create you own line by doing this:

```
(setq e '((0 . "LINE")(8 . "0")(10  50.0 50.0 0.0)(11 100.0 100.0 0.0)))
(entmake e)
```

Makes you think, doesn't it??

Cheers for now.....

Extended Entity Data

What is Extended Entity Data?
For years AutoCAD has had the ability to store user information within a drawing by utilising attributes. But, attributes have got their limitations.
They've got to be a block, or part of a block. They can be difficult to use.
They can be exploded or modified by the user and they can only hold certain types of information.

Extended Entity Data though, allows you to attach up to 16K of information to each and every entity in the drawing. You can also keep it totally separate from other information and, because it uses a uniquely registered name, can be kept relatively safe from being overwritten. You can also store lot's of different types of information in Extended Entity Data.

Extended Entity Data is attached to an entity as an associated list with a code number of -3. The simplest form of an Xdata Associative list would look something like this :

```
((-3 ("AFRALISP" (1000 . "Kenny is great"))))
```

Firstly, let's look at some of the different types of xdata that you can attach to an entity :

String	1000.	A string of up to 255 characters.
Application Name	1001.	An Application Name.
Layer Name	1003.	The name of a Layer.
DataBase Handle	1005.	The handle of an entity.
3D Point	1010.	A 3D Coordinate value.
Real	1040.	A real value.
Integer	1070.	A 16 bit integer (signed or unsigned).
Long	1071.	A 32 bit signed (long) integer.
Control String	1002.	A control code to set off nested list.
World Space Position	1011.	A 3D coordinate point that is moved, scaled rotated, streched and mirrored along with the entity.
World Space Displacement	1012.	A 3D coordinate point that is scaled, rotated or mirrored along with the entity. It cannot be stretched.
World Space Direction	1013.	A 3D coordinate point that is rotated or mirrored along with the entity. It cannot be scaled, streched or moved.
Distance	1041.	A real value that is scaled along with the entity. Used for distance.
Scale Factor	1042.	A real value that is scaled along with the entity. Used as a scale factor.

Another important thing to remember about Xdata is that you can have more than one of the same associative code.

Let's attach some xdata to an entity in a drawing. Draw a line then type this:

```
(regapp "AFRALISP")
```

AutoLisp should return :

```
"AFRALISP"
```

You have now registered your external entity data name. This name is a unique identifier to your own extended entity data.
Next we need to get the entity data list of the entity that we want to add exdata to. Type this :

```
        (setq oldlist (entget (car (entsel))))
```

AutoLisp should return something like this:

```
Select object: ((-1 . ) (0 . "LINE") (5 . "271") (100 .
"AcDbEntity") (67 . 0) (8 . "0") (100 . "AcDbLine") (10 336.561 591.45 0.0) (11
672.362 497.304 0.0) (210 0.0 0.0 1.0))
```

Now, let's create the exdata that we want to add to the entity:

```
(setq thedata '((-3 ("AFRALISP" (1000 . "Kenny is handsome")
(1000 . "And intelligent"))))
```

Append it to the entity data list:

```
(setq newlist (append oldlist thedata))
```

Now, update the entity:

```
(entmod newlist)
```

We have now attached the xdata to the entity. To retrieve it we would type this:

```
(setq elist (entget (car (ensel)) '("AFRALISP")))
```

This would return the modified entity list. It should look something like this:

```
Select object: ((-1 . ) (0 . "LINE") (5 . "271") (100 .
"AcDbEntity") (67 . 0) (8 . "0") (100 . "AcDbLine") (10 336.561 591.45 0.0) (11
672.362 497.304 0.0) (210 0.0 0.0 1.0) (-3 ("AFRALISP" (1000 . "Kenny is
handsome") (1000 . "And intelligent"))))
```

To retrieve the xdata we would type this:

```
(setq exlist (assoc -3 elist))
```

This gets the xdata list from the entity list.

```
(-3 ("AFRALISP" (1000 . "Kenny is handsome") (1000 . "And intelligent")))
```

To retrieve the xdata itself we would type this:

```
(setq thexdata (car (cdr exlist)))
```

Now, we should have this:

```
("AFRALISP" (1000 . "Kenny is handsome") (1000 . "And intelligent"))
```

We now have an ordinary list. Because we created the xdata list, and we know in what order we created it, it's very simple to retrieve each individual part:

```
(setq data1 (cdr (nth 1 thexdata)))
(setq data2 (cdr (nth 2 thexdata)))
```

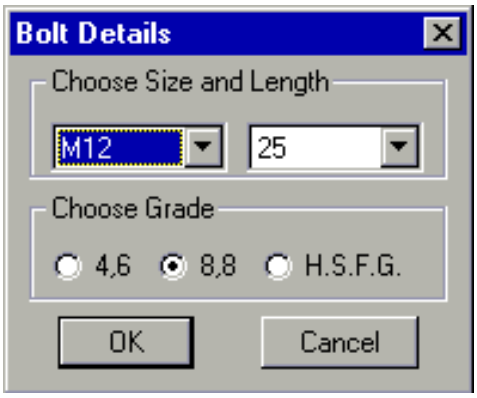
This should return:

```
"Kenny is handsome"
"And intelligent"
```

This, of course, is a factual statement.

Next we will have a look at a practical example using Xdata.
Come on, don't be scared.....

We are now going to write an application that attaches, as Extended Entity Data, bolt details to holes in a drawing. The details we will store will be Bolt Size, Bolt Length and Bolt Grade. We will design a dialogue box to enter this data.
If no data has been attached, the application will firstly attach some default data and then allow you to edit this. If data has been already attached, it will display this data and then allow you to change it, if you so wish.



Let's look at the Dialogue Box Coding first:

```
exbolt : dialog {                                   //dialog name
        label = "Bolt Details";                     //give it a label
        : boxed_row {                               //start boxed row
        label = "Choose Size and Length";           //give it a label
        : popup_list {                              //define list box
        key = "sel1";                               //give it a name
        }                                           //end list
        : popup_list {                              //define list box
        key = "sel2";                               //give it a name
        }                                           //end list
        }                                           //end boxed row
        : boxed_radio_row {                         //start boxed radio row
        label = "Choose Grade";                     //give it a label
        : radio_button {                            //define radio button
        key = "rb1";                                //give it a name
        label = "4,6";                              //give it a label
        }                                           //end radio button
        : radio_button {                            //define radio button
```

```
        key = "rb2";                                    //give it a name

        label = "8,8";                                  //give it a label

        }                                               //end radio button
        : radio_button {                                //define radio button

        key = "rb3";                                    //give it a name

        label = "H.S.F.G.";                             //give it a label

        }                                               //end radio button

        }                                               //end boxed radio row


        ok_cancel ;                                     //predifined OK/Cancel


}                                                       //end dialog
```

And now the AutoLisp code:

```
(defun c:exbolt ( / )
;define function

        (setvar "cmdecho" 0)
        ;switch off command echo

        (prompt "\nSelect the Hole to Add/Modify Xdata : ")
        ;prompt the user

        (setq e (entget (car (entsel)) '("AFRALISP")))
        ;get the associative code list

        (setq e1 (assoc -3 e))
        ;get the xdata

        (if (not e1)
        ;if there is no exdata

        (progn
        ;do the following

        (if (not (tblsearch "APPID" "AFRALISP"))
        ;check if the application has been registered

                (regapp "AFRALISP")
                ;if not, register it

        );if

                (setq e1 '((-3 ("AFRALISP"
                           (1000 . "3")
```

```
                    (1000 . "3")
                    (1000 . "8,8")
            ))))
            ;create a default xdata list

            (setq e (append e e1))
            ;append to to the main list

            (entmod e)
            ;modify the entity

      );progn

      );if

      (setq e2 (assoc -3 e))
      ;get the code -3 list

      (setq e3 (car (cdr e2)))
      ;get the exdata list

      (setq SIZ (cdr (nth 1 e3)))
      ;get the bolt size index number

      (setq SIZ1 (cdr (nth 2 e3)))
      ;get the bolt length index number

      (setq gr (cdr (nth 3 e3)))
      ;get the bolt grade

      (setq userclick T)
      ;set flag

      (setq NAMES '("M6" "M8" "M10" "M12" "M16" "M20" "M24" "M30" "M36"))
      ;create list of bolt sizes


      (setq LEN '("10" "15" "20" "25" "30" "35" "40" "45" "50" "55" "60"))
      ;create list of bolt lengths

      (setq dcl_id (load_dialog "exbolt.dcl"))
      ;load dialogue

      (if (not (new_dialog "exbolt" dcl_id)
      ;check for errors

            );not

           (exit)
           ;if problem exit

      );if

(set_tile "sel1" SIZ)
;initilise list box
```

```
(set_tile "sel2" SIZ1)
;initilise list box

(start_list "sel1")
;start the list

(mapcar 'add_list NAMES)
;add the bolt size

(end_list)
;end the list

(start_list "sel2")
;start the list

(mapcar 'add_list LEN)
;add the lengths

(end_list)
;end the list

(cond
;on condition

      ((= gr "4,6") (set_tile "rb1" "1"))
      ;if GR 4,6 switch on radio button rb1

      ((= gr "8,8") (set_tile "rb2" "1"))
      ;if GR 8,8 switch on radio button rb2

      ((= gr "HSFG") (set_tile "rb3" "1"))
      ;if GR HSFG switch on radio button rb3

);end cond

(action_tile "rb1"
;if radio button rb1 selected

      "(setq gr \"4,6\")"
      ;set grade of bolt

);action_tile

(action_tile "rb2"
;if radio button rb2 selected

      "(setq gr \"8,8\")"
      ;set grade of bolt

);action_tile

(action_tile "rb3"
;if radio button rb3 selected
```

```
      "(setq gr \"HSFG\")"
      ;set grade of bolt

);action_tile

  (action_tile "cancel"
  ;if cancel selected

      "(done_dialog)
      ;end dialog

       (setq userclick nil)"
       ;set flag to nill

  );action_tile
  ;if cancel set flag to nil

(action_tile "accept"

       "(setq siz (get_tile \"sel1\"))
       ;get the bolt size

       (setq siz1 (get_tile \"sel2\"))
       ;get the bolt length

       (done_dialog)
       ;end the dialog

       (setq userclick T)"
       ;set the flag to true

);action tile

(start_dialog)
;start the dialogue

(unload_dialog dcl_id)
;unload the dialogue

 (if userclick
 ;if OK has been selected

  (progn
  ;do the following

      (setq NSIZ (cons 1000 SIZ))
      ;construct a new bolt size list

      (setq NSIZ1 (cons 1000 SIZ1))
      ;construct a new bolt length list

      (setq NGR (cons 1000 gr))
      ;construct a new bolt grade list

      (setq e4 (chnitem NSIZ 2 e3))
```

```
        ;change the existing bolt size list

        (setq e5 (chnitem NSIZ1 3 e4))
        ;change the existing bolt length list

        (setq e6 (chnitem NGR 4 e5))
        ;change the existing bolt grade list

        (setq e7 (subst e6 e3 e2))
        ;update list

        (setq e8 (subst e7 e2 e))
        ;update list

        (entmod e8)
        ;update the entity

     (setq SIZ (nth (atoi SIZ) NAMES))
     ;get the size of the bolt from the list

     (setq SIZ1 (nth (atoi SIZ1) LEN))
     ;get the length of the bolt from the list

     (alert (strcat "The size of bolt is " SIZ "\n"
            "The length of bolt is " SIZ1 "\n"
            "The grade of bolt is " GR)
     );alert


    );end progn

   );end if

(princ)
;finish cleanly

);end defun


;;This function replaces any element in a list with another element
;;It requires 3 parameters (chnitem value itemnumber list)

(defun chnitem (value num lst)
     (setq num (- num 1))
     (setq tmplt (list nil))
     (setq tmplt2 (list nil))
     (setq counter 0)
     (repeat  num
          (setq tmplt (append tmplt (list (nth counter lst))))
          (setq counter (+ counter 1))
     )
     (setq counter (+ counter 1))
     (repeat (- (length lst) (+ num 1))
          (setq tmplt2 (append tmplt2 (list (nth counter lst))))
          (setq counter (+ counter 1))
```

```
    )
    (setq tmplt (cdr tmplt))
    (setq tmplt2 (cdr tmplt2))
    (setq lst (append tmplt (list value) tmplt2))
)
(princ)
;load cleanly
```

Draw a few circles representing holes on your drawing and run this application. Clever, hey!!

Now, let's write a routine that extracts this data from the drawing and saves it, in a space deliminating format, to an external text file.

```
(defun C:exfile ()
;define the function

  (setq NAMES '("M6" "M8" "M10" "M12" "M16" "M20" "M24" "M30" "M36"))
  ;create list of bolt sizes

  (setq LEN '("10" "15" "20" "25" "30" "35" "40" "45" "50" "55" "60"))
  ;create list of bolt lengths

  (setq fname (getstring "\nEnter file name: "))
  ;get the file name

  (setq fn (open fname "w"))
  ;open it to write

  (setq a (ssget "x" '((-3 ("AFRALISP")))))
  ;get all the bolt exdata in the drawing

  (setq lg (sslength a))
  ;get the number of bolts

  (setq i 0)
  ;set counter to zero

  (repeat lg
  ;repeat for the number of entities with bolt xdata

    (setq e (entget (ssname a i) '("AFRALISP")))
    ;get the entity list

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

    (setq d (assoc -3 e))
    ;get the xdata list

    (setq d1 (cdr (car (cdr d))))
    ;get just the xdata
```

```
    (mapcar 'set '(SIZ SIZ1 GR) d1)
    ;put each of the lists into variables

    (setq SIZ (cdr SIZ))
    ;get the index number of the bolt size

    (setq SIZ (nth (atoi SIZ) NAMES))
    ;retrieve the bolt size from the master list

    (setq SIZ1 (cdr SIZ1))
    ;get the index number of the bolt length

    (setq SIZ1 (nth (atoi SIZ1) LEN))
    ;retrieve the bolt size from the master list

    (setq GR (cdr GR))
    ;retrieve the bolt grade

    (princ SIZ fn)
    ;print bolt size

    (princ " " fn)
    ;print a space

    (princ SIZ1 fn)
    ;print bolt length

    (princ " " fn)
    ;print a space

    (princ GR fn)
    ;print the bolt grade

    (princ "\n" fn)
    ;print a new line

  );end repeat

  (close fn)
  ;close the file

  (alert (strcat "Exported Details of\n"
         (itoa lg)
         " Bolts to file : \n"
         (strcase fname))
  );alert
  ;inform the user

  (princ)
  ;finish cleanly

);end defun

(princ)
;load cleanly
```

As you can see, you can attach xdata to any type of entity, and attach all sorts of different types of information.

(mapcar) and (lambda).

As you know, LISP stands for "List Processing". There are quite a few commands in AutoLisp that allow you to manipulate lists. (subst, append, etc.)
But what about commands that will allow you to apply a function to items in a list. Let's look at (mapcar) first.

(mapcar).

The function (mapcar) allows you to perform a "function" on each element of the list.
Let's try a simple example :

What we want to do is add 1 to each element of a list.

```
(setq a 1)
(mapcar '1+ (list a))
```

This will return 2.
This is what happened :

(mapcar...adds 1 to the (list a) to make 2.

Now a longer list :

```
(setq a '(1 2 3 4 5 6 7 8 9 10))
(mapcar '1+ a)
```

This will return :

```
(2 3 4 5 6 7 8 9 10 11)
```

Just a few words on creating lists in AutoLisp.
There are two ways to create lists.
The first way is to use a command that will create a list.
Examples of this are (getpoint) and (entget).
Both of these commands will return a list.

Secondly, you could use the (list) command.
For example :

```
(setq a (list 1 2 3 4 5))
```

If you looked at variable a it would look like this :

```
(1 2 3 4 5)
```

The other way of writing this is :

```
(setq a '(1 2 3 4 5))
```

Both methods do the same thing, create a list.

Here is another example using (mapcar) :

Say you have a list of data stored in variable *arglist*

```
(setq arglist '(12.0 145.8 67.2 "M20"))
```

You want to place each item in the list in it's own variable to use in your routine. One way to do it would be as follows :

```
(setq a (nth 0 arglist))
(setq b (nth 1 arglist))
(setq c (nth 2 arglist))
(setq d (nth 3 arglist))
```

This works, but is an extremely slow way of processing the data as each variable requires a program statement.
A much more efficient way is to use the MAPCAR technique.

```
(mapcar 'set '(a b c d) arglist)
```

This routine maps the SET function to each element of the first list and it's corresponding element of the second list. SET is used instead of SETQ to evaluate each quoted element of the first list.
With the currently set list c, it sets a to 12.0, b to 145.8, c to 67.2 and d to "M20".

If you are reading a list from an external file, your routine may not read back the elements of the list as they once were. Your routine will read them back as strings. For example :

Your list should look like this :

```
(10 20 30 40 50)
```

But after reading the list in from the file, it looks like this :

```
("10" "20" "30" "40" "50")
```

You can use (mapcar) to convert the list from strings to integers :

```
(setq b (mapcar '(atoi) thelist))
```

Now this works fine if you are using an AutoLisp function, but how would you use (mapcar) with a user defined function?
Let's look at this example :

What we have is a list of angles that we want to convert to Radians.

```
(setq c '(23.0 47.8 52.1 35.6))
```

Firstly we would write a function to convert degrees to radians.

```
(defun dtr (a)
        (* pi (/ a 180.0))
)
```

Our function to convert our list would look like this :

```
(setq d (mapcar 'dtr c))
```

This function will run the (dtr) function against each element of list c.
In other words, the value of each element is passed to the (dtr) function.

The function could also be written like this :

```
(setq d (mapcar (quote dtr) c))
```

(lambda).

Now this is where the (lambda) function comes into play.
(lambda) allows you to write the (dtr) function "in-line" within the (mapcar) expression without having to define a separate function.

```
(setq d (mapcar (quote (lambda (a) (* pi (/ a 180.0)))) c))
```

or

```
(setq d (mapcar '(lambda (a) (* pi (/ a 180.0))) c))
```

This function will convert all the angles in the list c to radians and store them in variable

d.

Let's look a bit closer at the (lambda) function.

```
        (lambda (a) (* pi (/ a 180.0)))
```

is the same as

```
        (defun (a) (* pi (/ a 180.0)))
```

Let's write a function to test this :

```
(defun c:test ()
        (setq c '(23.0 47.8 52.1 35.6))
        (setq d (mapcar '(lambda (a) (* pi (/ a 180.0))) c))
        (mapcar 'set '(w x y z) d)
   (princ)
)
```

!c should return (23.0 47.8 52.1 35.6)
!d should return (0.401426 0.834267 0.909317 0.621337)
!w should return 0.401426
!x should return 0.834267
!y should return 0.909317
!z should return 0.621337

To quote the AutoCad Customization Manual :

"Use the (lambda) function when the overhead of defining a new function is not justified. It also makes the programmer's intention more apparent by laying out the function at the spot where it is to be used."

In practice, (lambda) can be used anywhere you need a speedy function and you don't want the trouble of writing and making sure a (defun) function is loaded.

There are another two AutoLisp commands that can apply a function to a list.
They are (apply) and (foreach).
Let's have a quick look at them.

(apply)

This function differs from (mapcar) in that it applies a function to the whole list and not to the individual items in the list.
Here's a couple of examples :

```
(apply '+ '(1 2 3))
```

This will return 6

```
(apply 'strcat '("a" "b" "c"))
```

This will return "abc"

You can also use (apply) in comjunction with (lambda) :

```
(apply '(lambda (x y z)
        (* x (- y z))
        )
        '(5 20 14)
)
```

This will return 30 (20-14*5=30)

(foreach)

The syntax for this is :
(foreach name list expression.....)

This function steps through list, assigning each element to name, and evaluates each expression for every element in the list.
For example :

```
(foreach n a (princ n) (terpri))
```

If you had a list :

```
(setq a (1 2 3 4 5))
```

The previous expression would print vertically :

```
1
2
3
4
```

Right, I don't know about you, but my brain is full.
Time for a couple of beers. Cheers for Now....

The 'Eval' Function.

'Eval' is another of the least seldom used, but very powerful, AutoLisp functions. It is defined in the AutoCad Customization Guide as follows :

Eval - Returns the result of evaluating an AutoLisp expression.
Syntax : (eval *expr*)

For example, have a look at the following code segment :

```
(eval (ascii "A"))
```

Which simply means, *Evaluate the expression (ascii "A").*
This should return "65", the ASCII character code for "A".

Fine, I understand that, but where would I use it? Here's a simple example.
Say we had a whole lot of distances that we had to measure and then come up with an accumulative total. We could measure each one, write it down, and then, manually add them up. Here's a better way.
We are now going to write an AutoLisp routine that allows us to measure each distance, and then return a total of all the distances. Here we will use the 'eval' function to perform the final total calculation for us.
Here's the coding :

```
;Program to measure non-sequential distances

(Defun C:ADIST ( / dst dstlst adist)
;define the function and declare variables

  (setq dstlst '(+ ))
  ;create list with plus

    (while
    ;while a return is not entered

        (setq dst (getdist "\nPick point or Return to exit: "))
        ;get the distance

        (setq dstlst (append dstlst (list dst)))
        ;append the distance to the list

        (setq adist (eval dstlst))
        ;calculate the running total

        (setq dst (rtos dst 2 2))
        ;convert to string

        (setq adist (rtos adist 2 2))
        ;convert to string
```

```
        (princ (strcat "\nDistance = " dst "  Accum Dist = " adist))
        ;display the distance and the running total

    );end while

        (prompt (strcat "\nTotal Distance = " adist))
        ;display the total distance

   (princ)
   ;clean finish

);defun

(princ)
;clean loading
```

We start of the function by defining a list that only contains the '+' function.
The heart of the routine is nested in a 'while' loop. As long as the user keeps on adding distances, the routine runs, but as soon as he hits a return, 'while' evaluates to nil and the loop is ended.
The first line within the loop simply gets the required distance. The second line appends the distances to the list. Thirdly, we evaluate the list adding all of the distances together. We then format the distance and the running total and display them to the user. Once the user has finished selecting the distances, we display the final total. Simple really!!

Hint - Have you ever thought of building a macro recorder similar to that used in Excel. By storing the users AutoCAD commands in a list along with the AutoLisp command function, you could use the 'eval' function to 'playback' the list of commands. For example :

```
        (setq lst '(command "circle" pt2 rad))
        (eval lst)
```

Here's another example of using the (eval) function, but this time we'll use it to replace the (cond) function.
The following coding is a small application that converts various values. It uses the (cond) function to determine which radio button was selected and then uses that information to run the relevant sub-routine.
First the DCL coding :

```
//DCL CODING STARTS HERE
conversion   : dialog {
```

```
            label = "Conversion" ;

        : radio_button {
          key = "rb1" ;
          label = "&Inches to Millimetres" ;
          value = "1" ;
        }

        : radio_button {
          key = "rb2" ;
          label = "&Millimetres to Inches" ;
        }

        : radio_button {
          key = "rb3" ;
          label = "M&iles to Kilometres" ;
          }

        : radio_button {
          key = "rb4" ;
          label = "&Kilometres to Miles" ;
        }

        : edit_box {
          key = "eb1";
          label = "Value";
          value = "1.0";
        }


        ok_cancel ;

        }
//DCL CODING ENDS HERE
```

Save this as "Conversion.dcl".
And now the AutoLisp Coding :

```
;AUTOLISP CODING STARTS HERE
(defun C:Conversion ()

  (setq retval "IM")

  (setq edval 1.0)

  (setq dcl_id (load_dialog "conversion.dcl"))

  (if (not (new_dialog "conversion" dcl_id)
  ;test for dialog
```

```
    );not

    (exit)
    ;exit if no dialog

  );if

  (mode_tile "eb1" 2)

  (action_tile "rb1" "(setq retval \"IM\")")
  (action_tile "rb2" "(setq retval \"MI\")")
  (action_tile "rb3" "(setq retval \"MK\")")
  (action_tile "rb4" "(setq retval \"KM\")")

  (action_tile
    "accept"
    (strcat
    "(progn (setq edval (atof (get_tile \"eb1\")))"
    "(done_dialog) (setq userclick T))")
  );action tile

    (action_tile
    "cancel"
    "(done_dialog) (setq userclick nil)"
    );action_tile

  (start_dialog)

  (unload_dialog dcl_id)

   (if userclick

    (cond

     ((= retval "IM") (IM edval))
     ((= retval "MI") (MI edval))
     ((= retval "MK") (MK edval))
     ((= retval "KM") (KM edval))

    );cond

  );if userclick

  (alert (strcat "Value = " (rtos ans)))

 (princ)

);defun

;-------------------------
(defun IM (val)
 (setq ans (* val 25.4))
```

```
)
;-------------------------
(defun MI (val)
 (setq ans (/ val 25.4))
)
;-------------------------
(defun MK (val)
 (setq ans (* val 1.609344))
)
;-------------------------
(defun KM (val)
 (setq ans (/ val 1.609344))
)
;-------------------------
(princ)
;AUTOLISP CODING ENDS HERE
```

Save this a "Conversion.lsp" then load and run it.
As you will see, it's just a simple little conversion routine.

Now replace these lines :

```
(cond

  ((= retval "IM") (IM edval))

  ((= retval "MI") (MI edval))

  ((= retval "MK") (MK edval))

  ((= retval "KM") (KM edval))

);cond
```

with this line :

```
((eval (read retval)) edval)
```

This one line replaces the whole of the (cond) section.

Cheers for now and happy evaluating.....

Redefining Commands.

Most AutoCAD users know that it is possible to undefine an AutoCAD command and then redefine it to do something else. But, very few people seem to use this very useful feature. I find it very handy to stop people changing system settings that divert from drawing office standards. Let's have a look at an example of this.
Say we had a block inserted into the drawing that we did not want to be exploded because it contains attributes. This is how we would go about using AutoLisp to undefine and then redefine the 'Explode' command.

First, we would create an AutoLisp file called 'Redefs.Lsp'.
Then we would add the following coding :

```
(command "UNDEFINE" "EXPLODE")
'undefine the Explode command
```

After undefining the explode command we would then redefine it:

```
(defun C:EXPLODE ( / lst1 en typ)

   (setq lst1 (list "DRGTITLE1" "DRGTITLE2"))
   ;list of block names that must NOT be exploded

   ;If you wish to add additional block names, and want an easy way to
   ;figure out which ones you screened for each release, use this line with
   ;your block names.
   ;(setq lst1 (append lst1 (list "DRGTITLE2" "DRGTITLE3" "DRGTITLE4")))

   (setq en (car (entsel "\n Select block reference,
                  polyline, dimension, or mesh: ")))
   ;gets the block and mimics the explode prompt

   (setq typ (entget en))
   ;get the entity data

   (setq typ (cdr (assoc 2 typ)))
   ;get the block name

   (if (member typ lst1)
       ;if the selected block name is a member of our list

         (alert "\nThis Block Cannot be Exploded.
                 \n    Refer to System Manager")
         ;inform the user

         ;if it is not
         (progn
         ;do the following

         (command ^c^c)
         ;cancel any commands

         (command ".EXPLODE" en)
```

```
        ;explode the block

      );progn

   )
   ;if

(princ)
;finish clean

);defun
(princ)
;load clean
```

We would, of course, load Redefs.Lsp from our Acad.Lsp file to ensure that it would be available all of the time. Just please note, that because this routine uses the 'Command' function, we would have to load it from the S::STARTUP section of the Acad.Lsp file. It would look something like this :

```
(defun S::STARTUP ()
       (prompt "\nAfraLisp Custom Utilities Loaded....\n")
       (load "REDEFS" "\nREDEFS.LSP not found")
    (princ)
);defun
```

I'll tell you what we'll do next. Let's create our own Drawing AutoSave.
First of all we need a way to check when a certain amount of time has elapsed. One of the simplest ways of doing this is to use one of the commonest used commands as a trigger. We'll use the 'Line' command.
Add the following coding to Redefs.Lsp :

Firstly we need to undefine the 'Line' command:

```
(command "UNDEFINE" "LINE")
;undefine the Line command
```

The we need to redefine it:

```
(defun C:LINE ()
;define the function

    (autosave)
    ;call the Autosave function

    (command ".LINE")
    ;call the line command

  (princ)
```

```
);defun
```

Next we need to write our Autosave function:

```
(defun AUTOSAVE ( / T1 ECC)
;define the function

      (setq ECC (getvar "CMDECHO"))
      ;get the value of the CMDECHO system variable

      (setvar "CMDECHO" 0)
      ;switch it off

      (if (not T3) (setq T3 (getvar "TDUSRTIMER")))
      ;check if we have the value of the drawing timer
      ;if we haven't got it, then get it.

      (if (not T2) (setq T2 (getreal "\nHow many minute between Saves ??: ")))
      ;check if we have an AutoSave time.
      ;if we haven't got it, then get it.

      (setq T1 (getvar "TDUSRTIMER"))
      ;get the drawing timer again for comparison purposes.

      (if (> (- T1 T3) (/ T2 60.0 24.0))
      ;compare the drawing timer values

      (progn
      ;if it is time to save

            (prompt "\nAutoSaving Drawing..Please Wait...... ")
            ;inform the user

            (command "QSAVE")
            ;save the drawing

            (setq T3 (getvar "TDUSRTIMER"))
            ;reset the timer

      );progn

      );if

      (setvar "CMDECHO" ECC)
      ;reset CMDECHO

   (princ)

);defun
```

The first time you select the Line command, this function will ask you for an interval between

saves. From then on, every time you use the Line command the function will check to see if you have exceeded the time interval. If you haven't, it will do nothing. But if you have, it will first inform you, and then save your drawing. Handy little thing, Hey...

Efficient Variables.

Every variable used in an AutoLisp routine uses up system resources. You've probably been advised before now, not to use too many SETQ's. LISP, itself was originally a functional language, without any variables. Here is a typical example of how we all go about collecting the values required for our routines.

```
(setq "oldecho" (getvar "CMDECHO"))
(setq "oldhigh" (getvar "HIGHLIGHT"))
(setq "oldsnap" (getvar "OSMODE"))
(setq pt1 (getpoint "\Enter First Point : "))
(setq pt2 (getpoint "\Enter Second Point : "))
(setq thk (getdist "\Enter Thickness : "))
(setq qty (getint "\Enter Number Required : "))
```

Programmatically, there is nothing wrong with this coding, except that seven (7) variables are used, each of which takes up system resources. As well as this, each variable will need to be declared as Local to prevent a scattering of loose variables all over the place.

A much better way of storing your variables is to place them in a list.
Here's one way of doing this :

```
(setq AList (append (list (getvar "CMDECHO")) AList))
(setq AList (append (list (getvar "HIGHLIGHT")) AList))
(setq AList (append (list (getvar "OSMODE")) AList))
(setq AList (append (list (getpoint "\Enter First Point : ")) AList))
(setq AList (append (list (getpoint "\Enter Second Point : ")) AList))
(setq AList (append (list (getdist "\Enter Thickness : ")) AList))
(setq AList (append (list (getint "\Enter Number Required : ")) AList))
```

If we ran this sequence of coding, the variable *AList* would contain something like this :

```
(5 10.0 (660.206 391.01 0.0) (411.014 548.932 0.0) 32 1 0)
```

Now this is a lot better. Every variable stored in one list and only one variable to declare. To retrieve any of the values, we would simply do the following :

```
(setvar "OSMODE" (nth 4 AList))
```

This, of course would return 32, the value of the Snap that we previously saved.

The problem with this though, is remembering which value belongs to what. To achieve this we would need to keep track of the order in which we passed the values to our list.
Wouldn't it be great if we could pass a label along with each value so that we could easily retrieve the value by just quoting the label.
By using *'Associative Lists'* we can do just that. Have a look at this :

```
(setq AList (append (list (cons "OLDECHO" (getvar "CMDECHO"))) AList))
(setq AList (append (list (cons "OLDHIGH" (getvar "HIGHLIGHT"))) AList))
(setq AList (append (list (cons "OLDSNAP" (getvar "OSMODE"))) AList))
(setq AList (append (list (cons "PT1" (getpoint "\Enter First Point : "))) AList))
(setq AList (append (list (cons "PT2" (getpoint "\Enter Second Point : "))) AList))
(setq AList (append (list (cons "THK" (getdist "\Enter Thickness : "))) AList))
(setq AList (append (list (cons "QTY" (getint "\Enter Number Required : "))) AList))
```

This coding would return something like this :

```
(("QTY" . 6) ("THK" . 12.0) ("PT2" 809.113 523.118 0.0) ("PT1" 356.314 646.115 0.0)
("OLDSNAP" . 32) ("OLDHIGH" . 1) "OLDECHO" 0)
```

Now, to retrieve any value from this list, irrespective of it's position in the list, we would simply do something like this :

```
        (setvar "OSMODE" (cdr (assoc "OLDSNAP" AList)))
```

This, again would return 32.

Now, by converting the construction of the list, and the retrieving of values from the list, into functions, we have an elegant and sophisticated way of storing our variables in an efficient and practical manner. Following, is an AutoLisp routine that can be used, with just a wee bit of modification, in any situation to make the storage of your variable much more efficient :

```
(defun c:efflist ( / AnItem item MainList)

        (setq AnItem (getvar "OSMODE"))
        ;get the snap setting

        (AList "OLDSNAP" AnItem)
        ;add it to the list

        (setq AnItem (getvar "HIGHLIGHT"))
        ;get the highlight setting

        (AList "OLDHIGH" AnItem)
        ;add it to the list

        (setq AnItem (getvar "CMDECHO"))
        ;get the command echo setting

        (AList "OLDECHO" AnItem)
        ;add it to the list

        (setvar "Osmode" 32)
        ;reset snap to intersection

        (setvar "Highlight" 0)
        ;switch off highlight

        (setvar "Cmdecho" 0)
        ;switch off command echo

        (setq AnItem (getpoint "\nSelect First Point : "))
        ;get the first point

        (AList "FirstPoint" AnItem)
        ;add it to the list

        (setq AnItem (getpoint AnItem "\nSelect Second Point : "))
        ;get the second point

        (AList "SecondPoint" AnItem)
        ;add it to the list

        (setq AnItem (getpoint AnItem "\nSelect Third Point : "))
```

```
        ;get the third point

        (AList "ThirdPoint" AnItem)
        ;add it to the list

        (setq AnItem (getpoint AnItem "\nSelect Fourth Point : "))
        ;get the fourth point

        (AList "FourthPoint" AnItem)
        ;add it to the list

        (command "Line" (RList "FirstPoint")
                        (RList "SecondPoint")
                        (RList "ThirdPoint")
                        (RList "FourthPoint")
                        "C"
        )
        ;retrieve all the point values and draw the shape

        (setvar "OSMODE" (RList "OLDSNAP"))
        ;retrieve and reset snap

        (setvar "HIGHLIGHT" (RList "OLDHIGH"))
        ;retrieve and reset highlight

        (setvar "CMDECHO" (RList "OLDECHO"))
        ;retrieve and reset command echo

  (princ)

);defun

;This function constructs the list and adds it to the main list
(defun AList (Name Val)

        (setq item (list (cons Name Val)))
        ;construct list

        (setq MainList (append item Mainlist))
        ;add it to the main list

);defun

;This function retrieves the values from the main list
(defun RList (TheName)

        (cdr (assoc TheName MainList))
        ;retrieve value from list

);defun
(princ)
```

After running this routine, the value of the variable *MainList* would look something like this :

```
(("FourthPoint" 456.598 514.007 0.0) ("ThirdPoint" 676.92 293.827 0.0)
("SecondPoint" 1030.95 526.155 0.0) ("FirstPoint" 576.636 732.669 0.0)
("OLDECHO" . 0) ("OLDHIGH" . 1) ("OLDSNAP" . 32))
```

The "Dir" Command

The "DIR" command is a DOS (remember that dark place) command that displays a list of a directories files and subdirectories. But did you know that you can use the "DIR" command from within AutoLisp? This is a great function for obtaining a list of files in a directory, especially useful for Batch Processing Routines.
Let's have a look at it. Fire up AutoCAD and type this at the Command prompt :

```
Command: (command "dir" "*.* > Temp.txt")
```

What this statement is asking for is a directory listing of all the files in the current directory output to the file Temp.txt.
Now open Temp.txt and you should have something like this :

```
 Volume in drive O has no label
 Directory of O:\E51D\E51D1

.           <DIR>     09-11-99  3:13p .
..          <DIR>     09-11-99  3:13p ..
9961     DXF       345,167  08-05-99 12:14p 9961.DXF
9962     DXF     8,246,298  08-05-99 12:14p 9962.DXF
Ant      exe       701,121  08-05-99 11:36a Ant.exe
batchp1  exe        11,776  01-18-99 11:23a batchp1.exe
BATCHP1 EXE        13,312   01-19-99  8:19a BatchPurge.Exe
DCLTUT   GIF         2,635  08-01-99 11:19p DCLTUT.GIF
debex       <DIR>     07-28-99  7:04a debex
N1       dxf       245,702  08-02-99  3:23p N1.dxf
NAMDEB   DXF    28,320,177  07-30-99 12:29p NAMDEB.DXF
projair  dwg       120,055  07-06-99  3:11p projair.dwg
R12K471 DWG       421,118   07-02-99 10:28a R12K47646.dwg
sort     dwg       125,471  07-26-99  1:17p sort.dwg
SORTER   dwg        27,981  07-27-99 10:35a SORTER.dwg
temp        <DIR>     09-11-99  3:13p temp
temp     txt             0  09-11-99  3:13p temp.txt
test       <DIR>     08-20-99 11:15a test
title    dwg        99,381  08-02-99  2:30p title.dwg
truss1   dwg        84,382  07-09-99 11:13a truss1.dwg
truss2   dwg        88,296  07-09-99 11:13a truss2.dwg
uniglide dwg       205,715  07-19-99  9:09a uniglide.dwg
VALVE    dwg        24,693  07-29-99 11:31a VALVE.dwg
        18 file(s)     39,083,280 bytes
         5 dir(s)   2,147,450,880 bytes free
```

This is a listing of all the files and sub directories contained in your current working directory.
O.K. I know, it's not quite formatted the way we would want it to be.
So, let's use a couple "DIR" switches to format the file list the way that we would like it.

Try this :

```
Command: (command "dir" " /b *.* >temp.txt")
```

That's better, we've now got just the file names. The /b switch limits the list to only file names. Temp.txt should now look something like this :

```
9961.DXF
9962.DXF
Ant.exe
batchp1.exe
BatchPurge.Exe
DCLTUT.GIF
debex
N1.dxf
NAMDEB.DXF
projair.dwg
R12K47646.dwg
sort.dwg
SORTER.dwg
temp
temp.txt
test
title.dwg
truss1.dwg
truss2.dwg
uniglide.dwg
VALVE.dwg
```

Right, now retrieve just the DWG filenames :

```
Command: (command "dir" " /b *.dwg > temp.txt")
```

Temp.txt should now look something like this :

```
projair.dwg
R12K47646.dwg
sort.dwg
SORTER.dwg
title.dwg
truss1.dwg
truss2.dwg
uniglide.dwg
VALVE.dwg
```

That's it, we have now got our required file listing in the format that we require. The "DIR" command has got a lot more switches that you can use to restrict your listings to

whatever you desire. I suggest you dust off one of your old DOS Reference Manuals and look "DIR" up.

Hint :

Would you like a listing of all the files in the current directory AND all sub-directories? Use the /s switch :

```
Command: (command "dir" " /b/s *.dwg >temp.txt")
```

You should get a listing something like this :

```
O:\E51D\E51D1\projair.dwg
O:\E51D\E51D1\R12K47646.dwg
O:\E51D\E51D1\sort.dwg
O:\E51D\E51D1\SORTER.dwg
O:\E51D\E51D1\title.dwg
O:\E51D\E51D1\truss1.dwg
O:\E51D\E51D1\truss2.dwg
O:\E51D\E51D1\uniglide.dwg
O:\E51D\E51D1\VALVE.dwg
O:\E51D\E51D1\test\SCANNEX1.dwg
O:\E51D\E51D1\test\SCANNEX2.dwg
O:\E51D\E51D1\test\SCANNEX3.dwg
O:\E51D\E51D1\test\SCANNEX4.dwg
```

Colours and Linetypes Bylayer.

Each layer in an AutoCAD drawing has a default colour and a default linetype. Thus, the term "Bylayer" means that an object drawn on a particular layer has the default colour and linetype of that layer.

You can, if you wish, change the colour and linetype of an object drawn on a layer to suit your requirements. A good example of this is when dimensioning. Your dimensions reside on say layer "Dim" which has a default color of "1" and a default linetype of "Continuous". The extension lines though, are also on layer "Dim" but have a color of say "9". When you plot, colour "1" has a pen width 0f 0.35mm but colour "9" has a pen width of 0.15mm which will plot out the extension lines thinner.

Often though, you have to deal with a total mismatch of Layers, Colours and Linetypes and would like to convert all objects back to their "Bylayer" default values. Let's have a look at how we would go about this.

First of all, open a new, blank drawing and setup a Layer with the following properties :

Layer Name "2"
Layer Colour 2 - "Yellow"
Layer Linetype Dashed2

Now, draw a line on Layer "2" anywhere in the drawing then enter the following :

Command: (setq sel (ssget)) Select the Line
Select objects: 1 found
Select objects: <enter>
<Selection set: 8>

Command: (setq entity (ssname sel 0))
<Entity name: 1838e10>

Command: (setq name (entget entity))
((-1 . <Entity name: 1838e10>) (0 . "LINE") (330 . <Entity name: 18388c0>) (5 . "95A") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (100 . "AcDbLine") (10 319.65 364.538 0.0) (11 338.809 500.026 0.0) (210 0.0 0.0 1.0))

We have just retrieved the entity list of the line object.
The DXF Group Code 8, is the layer name. Group Code 62 is the colour of the object, but where is it?

Let's have a look for it. Type this :

Command: (setq layer (cdr (assoc 62 name)))
nil

It returns "nil". Hey Kenny, what's going here?
If an object is drawn on a layer using colour "Bylayer", no Group Code 62 exists as

AutoCAD already knows what colour to draw the object with.
O.K. let's mess AutoCAD around and change the colour :

Command: change
Select objects: 1 found
Select objects: <enter>
Specify change point or [Properties]: p
Enter property to change [Color/Elev/LAyer/LType/ltScale/LWeight/Thickness]: c
Enter new color <ByLayer>: 4
Enter property to change [Color/Elev/LAyer/LType/ltScale/LWeight/Thickness]: <enter>

Ha, take that you bounder. Let's look at the entity list again :

Command: (setq sel (ssget))
Select objects: 1 found
Select objects: <enter>
<Selection set: a>

Command: (setq entity (ssname sel 0))
<Entity name: 1838e10>

Command: (setq name (entget entity))
((-1 . <Entity name: 1838e10>) (0 . "LINE") (330 . <Entity name: 18388c0>) (5 . "95A") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (62 . 4) (100 . "AcDbLine") (10 319.65 364.538 0.0) (11 338.809 500.026 0.0) (210 0.0 0.0 1.0))

Hey, Group Code 62 has suddenly appeared in our list and tells us that the object is now colour 4!!!
Jeepers, AutoCAD is only clever.

But what happens if we want to change the line back to it's "Bylayer" colour and Linetype?
Have a look at the following :

Select the object :
Command: (setq sel (ssget))
Select objects: 1 found
Select objects: <enter>
<Selection set: 4>

Get it's name :
Command: (setq entity (ssname sel 0))
<Entity name: 1803610>

Retrieve the entity list :
Command: (setq name (entget entity))
((-1 . <Entity name: 1803610>) (0 . "LINE") (330 . <Entity name: 18030c0>) (5 . "95A") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (62 . 4) (100 . "AcDbLine") (10 275.439 544.206 0.0) (11 357.967 544.206 0.0) (210 0.0 0.0 1.0))

Get the layer name :
Command: (setq layer (cdr (assoc 8 name)))

"2"

Use "*tblsearch*" to get the layer default details :
Command: (setq layerinf (tblsearch "LAYER" layer))
((0 . "LAYER") (2 . "2") (70 . 0) (62 . 2) (6 . "DASHED2"))

Retrieve the "Bylayer" colour :
Command: (setq layercol (cdr (assoc 62 layerinf)))
2

Construct a new list and append it to the entity list
Command: (setq name (append name (list (cons 62 layercol))))
((-1 . <Entity name: 1803610>) (0 . "LINE") (330 . <Entity name: 18030c0>) (5 . "95A") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (62 . 4) (100 . "AcDbLine") (10 275.439 544.206 0.0) (11 357.967 544.206 0.0) (210 0.0 0.0 1.0) (62 . 2))

Now modify the definition data of an object :
Command: (entmod name)
((-1 . <Entity name: 1803610>) (0 . "LINE") (330 . <Entity name: 18030c0>) (5 . "95A") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (62 . 4) (100 . "AcDbLine") (10 275.439 544.206 0.0) (11 357.967 544.206 0.0) (210 0.0 0.0 1.0) (62 . 2))

And finally, update the object :
Command: (entupd entity)
<Entity name: 1803610>

Your line should have changed back to colour Yellow.

Let's check if our entity list has been change :

Command: (setq sel (ssget))
Select objects: 1 found
Select objects: <enter>
<Selection set: 6>

Command: (setq entity (ssname sel 0))
<Entity name: 1803610>

Command: (setq name (entget entity))
((-1 . <Entity name: 1803610>) (0 . "LINE") (330 . <Entity name: 18030c0>) (5 . "95A") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (62 . 2) (100 . "AcDbLine") (10 275.439 544.206 0.0) (11 357.967 544.206 0.0) (210 0.0 0.0 1.0))

Yep, everything is hunky dory! Our entity list has been updated with the correct values.

Hey, this is great. Let's try the same thing, but this time with Linetypes.
Change your line to Linetype "CENTER2".

Command: CHANGE
Select objects: 1 found
Select objects: <enter>
Specify change point or [Properties]: P
Enter property to change [Color/Elev/LAyer/LType/ltScale/LWeight/Thickness]: LT
Enter new linetype name <ByLayer>: CENTER2
Enter property to change [Color/Elev/LAyer/LType/ltScale/LWeight/Thickness]: <enter>

Now let's change it back to it's "Bylayer" Linetype :

Command: (setq sel (ssget))
Select objects: 1 found
Select objects: <enter>
<Selection set: 4>

Command: (setq entity (ssname sel 0))
<Entity name: 17db268>

Command: (setq name (entget entity))
((-1 . <Entity name: 17db268>) (0 . "LINE") (330 . <Entity name: 17dacc0>) (5 .
"965") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (62 . 2) (6 ."CENTER2")
(100 . "AcDbLine") (10 170.804 532.425 0.0) (11 237.122 628.15 0.0)
(210 0.0 0.0 1.0))

Command: (setq layer (cdr (assoc 8 name)))
"2"

Command: (setq layerinf (tblsearch "LAYER" layer))
((0 . "LAYER") (2 . "2") (70 . 0) (62 . 2) (6 . "DASHED2"))

This time select Group Code 6 for the default Linetype:

Command: (setq layerltype (cdr (assoc 6 layerinf)))
"DASHED2"

Command: (setq name (append name (list (cons 6 layerltype))))
((-1 . <Entity name: 17db268>) (0 . "LINE") (330 . <Entity name: 17dacc0>) (5 .
"965") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (62 . 2) (6 ."CENTER2")
(100 . "AcDbLine") (10 170.804 532.425 0.0) (11 237.122 628.15 0.0)
(210 0.0 0.0 1.0) (6 . "DASHED2"))

Command: (entmod name)
((-1 . <Entity name: 17db268>) (0 . "LINE") (330 . <Entity name: 17dacc0>) (5 .
"965") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "2") (62 . 2) (6 ."CENTER2")
(100 . "AcDbLine") (10 170.804 532.425 0.0) (11 237.122 628.15 0.0)
(210 0.0 0.0 1.0) (6 . "DASHED2"))

Command: (entupd entity)
<Entity name: 17db268>

Now, is that not interesting? Let's put it into practice. Open Notepad and paste this coding into it, saving it as "ColorToLayer.Lsp" :

;CODING BEGINS HERE

```
(defun c:ColorToLayer ()

;clear the loop control variables
(setq i 0 n 0)

;prompt the user
(prompt "\n Select entities to analyze ")

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

;get the number of objects
(setq n (sslength sel))

   ;start the loop
   (repeat n

      ;get the entity name
      (setq entity (ssname sel i))

      ;now get the entity list
      (setq name (entget entity))

      ;if not Bylayer
      (if (not (assoc 6 name))

         ;do the following
         (progn

            ;retrieve the layer name
            (setq layer (cdr (assoc 8 name)))

            ;get the layer data
            (setq layerinf (tblsearch "LAYER" layer))

            ;extract the default layer colour
            (setq layercol (cdr (assoc 62 layerinf)))

            ;construct an append the new list
            (setq name (append name (list (cons 62 layercol))))

            ;update the entity
            (entmod name)

            ;update the screen
            (entupd entity)

         );progn

      );if


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

;loop
);repeat

(princ)

);defun
```

```
(princ)

;CODING END HERE
```

You can easily write coding that will do exactly the same for Linetypes and even incorporate the two if you wish.
You could even take it a step further and develop a fully fledged Layer Mapping Application Manager. Makes you think doesn't it?

Debugging

Screams, intermittent yells and bangs, kid's and dogs cowering in the corner. What is going on? Have I gone nuts? Have I finally cracked and lost my mind?
No, it's just me trying to debug my latest program. Go on laugh, but I'll guarantee we've all done it, and we'll all do it again.

Debugging can be a real pain. "But it worked just now!" "Why did it crash?" "The coding looks perfect. I've checked it line by line 23 times." "There must be something wrong with my computer."

This tutorial will start off by showing you some of the more common AutoLisp bugs that I've come across, and then show you a couple of things to assist you in your debugging efforts. Let's start with some common bugs.

The simplest and most common is yourself. I don't know if you are aware of the fact that most programming errors occur between the seat and the keyboard. Yes, sad but true. So, how do we fix you up? A couple of tips :

- Don't program or debug when you're tired. Walk away from it. Have a break, go shopping. You'll be amazed at how a break from a program clears the mind.
- Very rarely will it be a computer or system bug. 99.99999 times out of a hundred, it's YOU.
- The computer or any other sinister source will not change your program coding or the variable values therein. YOU DO.

The second type of bug is the *syntax* bug. A syntax bug is where the command is in error. For example when you've miss-spelt an AutoLisp function or command. e.g. *(setq ang1 (angl pt1 pt2))*
This is normally quite easy to spot and rectify as AutoLisp will not recognise the command and your program will stop.

Another VERY common syntax bug is naming variables after an AutoLisp function or command. The most common? Angle and T (*angle* and *t*). I must have seen these two used as variables a thousand times. Length (*length*) is another very common one.

The third type of bug is the *logical* bug. Now this little beastie has numerous faces and comes in all shapes and forms.
The simplest form is when you pass incorrect information to a function.

(setq ang (angle pt1 pt2))

But what happens if pt1 is *nil*. Crash!!
Another good example is trying to pass string values to a function that expects numbers.

(and vice-a-versa.)
Again, Crash!! Check your variable values and data types.
To check if a variable is a list, you can use the *(type)* command, which returns a variable type. But since a list is not a string you must test it :

```
( if ( = ( eval v ) LIST) . . .
```

You could also use *(listp)* to see whether a variable is a list :

```
( if ( = T ( listp a ) ) . . .
```

***(listp)* returns T if it is a list and nil if it isn't.**

To test whether a variable is a number, use the same test as earlier but test for REAL or INT. You can also use *(numberp)*, which works the same as *(listp)* on lists. If the variable is a real number or an integer, it returns T. If not, it returns nil.
To test for a positive or negative number use *(minusp)*. This returns T if a number is negative and nil if it is positive.

Oh, and before I forget, another good example of wrong data types is confusing radians with degrees. AutoLisp functions use radians, AutoCAD commands expect degrees.

Loops are another area that seem extremely susceptible to attracting bugs. Have a look at this :

```
(while flag
    (+ cntr 1)
    (program statements here)
      ( if ( = cntr 5)
         (setq flag nil)
      );if
);while
```

What is supposed to happen here is that the program will loop until *flag* is set to *nil*. *cntr* is supposed to be incremented by one at each loop. But, as you can see from this coding, (at least I hope you can see), *cntr* will never reach the value of 5. It will always be equal to 1. The correct coding should be :

```
(setq cntr (+ cntr 1))
```

System variables can also create untold problems, especially Snaps. Switch them On only when you need them and Off once finished or you'll find yourself snapping to some weird and wonderful parts of your drawing.

And please remember the golden rule. "Reset the system back to the way you found it." Before changing system variables, save their existing values then reset them back at the end of your program. And don't forget to to include these settings in your error traps so that if something untoward does happen, your error trap will reset them back to their original values.

Now let's have a quick look at some DCL coding bugs.

Have a look at the first one :

(action_tile "cancel" "(done_dialog) (setq bryt 1) (exit)")

I was told that every time the user clicked the "Cancel" button, AutoCAD would freeze. Any idea why? The second example has a similar error :

(action_tile

"accept"

"(progn

(setq rimel (get_tile \"ebrim\"))

(setq maxel (get_tile \"ebmax\"))

(done_dialog) (setq flag T) (cutt))"

)

The answer to both examples is that they were trying to call a function (exit) and (cutt) from within a dialog.
In other words, the dialog was still open when they called the function.
An easy error to make and one to look out for.
Please everybody, be careful out there!
Now we'll look at a couple of handy ideas for making your debugging a little be easier.

When writing an AutoLisp routine it's quite handy to have the program stop if it encounters an error. It's also nice to be able to view the variables whilst the program is running to ensure that they contain the correct values.

In Visual Lisp and Visual Basic you can insert "*breakpoints*" into your coding that do just that and use the Watch window to check on their values. If you are not using Visual Lisp or Visual Basic this can still easily be accomplished. Place the following statement within your coding where you would like your program to stop :

(setq bp (getstring))

This statement will stop your program and only continue once you hit the space bar.

To print the variables, put a *(princ)* statement to print the variable before the break point.

(princ variablename)

(setq bp (getstring))

Do remember to remove your print statements and breakpoints on completion of debugging.

You could if you wish turn this little trick into a sub function. This would be quite handy if you have a large program and can foresee a lot of debugging. Have a look at this small AutoLisp routine that changes an objects colour to "Bylayer."

```
;CODING BEGINS HERE

(defun c:clay ()

  ;clear the loop control variables
  (setq i 0 n 0)

  ;prompt the user
  (prompt "\n Select entities to analyze ")

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

  ;get the number of objects
  (setq n (sslength sel))

  ;start the loop
  (repeat n

        ;get the entity name
        (setq entity (ssname sel i))

        ;now get the entity list
        (setq name (entget entity))

        ;retrieve the layer name
        (setq layern (cdr (assoc 8 name)))

        ;get the layer data
        (setq layerinf (tblsearch "LAYER" layern))

        ;extract the default layer colour
        (setq layercol (cdr (assoc 62 layerinf)))
```

```
        (bpt)

        (setq bp (getstring))

        ;construct an append the new list
        (setq name (append name (list (cons 62 layercol))))

        ;update the entity
        (entmod name)

        ;update the screen
        (entupd entity)

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

  ;loop
  );repeat

        (bpt)

        (setq bp (getstring))

(princ)

);defun

(defun bpt ()

        (princ i)
        (princ "\n")
        (princ n)
        (princ "\n")
        (princ sel)
        (princ "\n")
        (princ layern)
        (princ "\n")
        (princ layercol)
        (princ "\n")
        (princ i)

);defun

(princ)

;CODING END HERE
```

I included a sub routine that simply stops the program and lists the values of selected variables at two places, one within the loop and one at the end of the program. This way you can easily track the value of the variables whilst your program is in progress. You

could even include this type of breakpoint function within your error trap whilst debugging.

Another good trick is to always make your variables "global" at the beginning of a program. If you make them "local", they may have no value at the end of the program and you won't be able to see what was in them. The same as for the breakpoints, remember to declare your variables as "local" after debugging.

Leaving your variables "global" can though, create it's own problems whilst debugging. You run various programs that use the same variable names and suddenly you find your variables "tripping" all over themselves. This can be difficult to find as sometimes your program runs perfectly well and the next minute Crash!! Look out for this one.

I also found this little tit bit at AcadX, which I'm sure they won't mind me sharing with you.
Prior to A2K, AutoLisp would dump a bug trace to the command line when an error was encountered.
While developers went out of their way to suppress this behavior in a released product, it was a handy tool during the debugging cycle. To get that same behavior in A2K, replace the standard error handler with this one:

```
(defun errdump (s)
  (vl-bt)
  (princ)
)
```

Oh, and one more thing! Please remember the power of your eyes. Watch the screen as your program runs. You can pick up a lot of clues from watching what happens to entities whilst your program runs.

Well, that's it for debugging. I hope this helped you and didn't leave you even more frustrated than before.
Remember, test, test, test and test.

Dictionaries and XRecords.

Written by Stig Madsen.
(Published here on AfraLisp with kind permission)

Since the dawn of times we've had at least 10 options to save private data with an AutoCAD drawing. Those were and are the system variables USERI1-5 and USERR1-5 that holds integers and real numbers, respectively. The idea is OK but the variables are exposed for everyone to use and you shouldn't assume that your private data will stay intact.
Somewhere around release 10 came XData (Extended Entity Data) which is a rather clever invention. With XData you can attach intelligent information to entities, and it works flawlessly. There aren't really drawbacks to the technique, but there are limitations that has to do with the amount and type of data you can attach.

During release 13 new forms of entities hit the deck in order to keep and maintain all sorts of data. Among those were Dictionaries and XRecords. Like XData, Dictionaries can be attached to any kind of entity and you can even attach XData to them. Additionally, with Dictionaries/Xrecords you can tell the drawing itself to keep your data because the drawing maintains a dictionary in which you can save all the data you want.

So what is a Dictionary and how does it work? Instead of lengthy explanations, it'll be much easier to look at one of the many Dictionaries that AutoCAD itself uses. Later we''ll briefly look at how to make our own simple Dictionary, add an Xrecord to it and save them both with the drawing.

So, fire up AutoCAD and pay attention. As mentioned, the drawing maintains a Dictionary that is always present. This is known as the 'named object dictionary' (although I prefer the term 'main dictionary'). In VBA lingo it's a collection object and like all other collections it just holds a series of other objects. In VBA it is accessed through the document object and in AutoLISP it's accessed by one function only, NAMEDOBJDICT:

Command: (setq mainDict (namedobjdict))
<Entity name: 16a9860>

Command: (entget mainDict)
((-1 . <Entity name: 16a9860>) (0 . "DICTIONARY")
(330 . <Entity name: 0>) (5 . "C") (100 . "AcDbDictionary")
(280 . 0) (281 . 1) (3 . "ACAD_GROUP")(350 . <Entity name: 16a9868>)
(3 . "ACAD_LAYOUT") (350 . <Entity name: 16a98d0>)
(3 . "ACAD_MLINESTYLE") (350 . <Entity name: 16a98b8>)
(3 . "ACAD_PLOTSETTINGS") (350 . <Entity name: 16a98c8>)
(3 . "ACAD_PLOTSTYLENAME") (350 . <Entity name: 16a9870>))

By looking at the entity list of the main Dictionary, the most important thing about Dictionaries becomes clear: they are complete entities by themselves. They are not fragments of data attached to other entities like XData are.

However, the Dictionary itself is not where you will store your raw data, - it is merely a container for other objects that in turn can hold the data. The main dictionary above shows 5 such objects. Its components are given by a unique name in group 3. Corresponding entity names are given by groups 350 that follow, but when referencing an object in a Dictionary it should be done by name only. For example, to reference the "ACAD_MLINESTYLE" object, use DICTSEARCH:

Command: (setq mlineDict (dictsearch (namedobjdict) "ACAD_MLINESTYLE"))
((-1 . <Entity name: 16a98b8>) (0 . "DICTIONARY") (5 . "17")

(102 . "{ACAD_REACTORS") (330 . <Entity name: 16a9860>) (102 . "}")
(330 . <Entity name: 16a9860>) (100 . "AcDbDictionary") (280 . 0)
(281 . 1) (3 . "Standard") (350 . <Entity name: 16a98c0>))

This member of the main Dictionary is a Dictionary itself. It holds all the mline styles that are available. Any style that is created with MLSTYLE is added to the "ACAD_MLINESTYLE" Dictionary. To explore a specific style we have to dig deeper and because we are dealing with Dictionaries we can use DICTSEARCH again - this time by searching the recently returned Dictionary:

Command: (setq mlineStd (dictsearch (cdr (assoc -1 mlineDict)) "Standard"))
((-1 . <Entity name: 16a98c0>) (0 . "MLINESTYLE") (5 . "18")
(102 . "{ACAD_REACTORS") (330 . <Entity name: 16a98b8>) (102 . "}")
(330 . <Entity name: 16a98b8>) (100 . "AcDbMlineStyle") (2 . "STANDARD")
(70 . 0) (3 . "") (62 . 256) (51 . 1.5708) (52 . 1.5708) (71 . 2) (49 . 0.5)
(62 . 256) (6 . "BYLAYER") (49 . -0.5) (62 . 256) (6 . "BYLAYER"))

Now we are getting somewhere! All properties of the mline style "Standard" are exposed in all their glory. Feel free to look up all properties in the DXF Reference. Want to change the color of multilines? Just SUBSTitute group 62 and ENTMOD the style as usual:

Command: (entmod (subst (cons 62 2)(assoc 62 mlineStd) mlineStd))
((-1 . <Entity name: 16a98c0>) ..etc.. (62 . 2) ..etc.. (62 . 2) ..etc..)

Ok, so a Dictionary is a container that can hold a number of objects. Why not use existing structures like symbol tables instead of complicating things? Many reasons, but two reasons come to mind. Symbol tables are maintained by the people who implemented them and to expand them to hold every possible custom object is not an option. Secondly, Dictionaries can be customized in a way that is not possible with symbol tables, and that opens a range of possibilities only limited by imagination.

Dictionaries and XRecords go hand in hand. Like Dictionaries, XRecords are handled as named objects and can be manipulated by the same functions that handle Dictionaries. In the following, we'll try to add our own Dictionary to the main dictionary. We will also create an XRecord to hold various informations and add it to our Dictionary.

When dealing with Dictionaries, at one point you will have to consider ownership. Which object is going to own the Dictionary? Will it hold generic data for your application or will it hold data that is specific for some entity or entities? In the first case you will probably use the main dictionary to save your data with the drawing. If your application is maintaining data for linetypes, you will probably add an extension dictionary to the linetype symbol table. Whatever the ownership, the Dictionary is initially created without ownership and for that purpose we'll use the function ENTMAKEX. It works like ENTMAKE, but it creates the entity without an owner - and it returns an entity name instead of an entity list. Let's make a function that adds our own Dictionary to the main dictionary. In this example we will name it "OUR_DICT":

```
(defun get-or-create-Dict (/ adict)

  ;;test if "OUR_DICT" is already present in the main dictionary
  (if (not (setq adict (dictsearch (namedobjdict) "OUR_DICT")))

    ;;if not present then create a new one and set the main
    ;; dictionary as owner
    (progn

      (setq adict (entmakex '((0 . "DICTIONARY")(100 . "AcDbDictionary"))))

      ;;if succesfully created, add it to the main dictionary
      (if adict (setq adict (dictadd (namedobjdict) "OUR_DICT" adict)))
    )

    ;;if present then just return its entity name
    (setq adict (cdr (assoc -1 adict)))

  )

)
```

If you want to see what happens to the dictionary when added to an owner, then stop the routine right after ENTMAKEX and use ENTGET to investigate the newly created entity. Notice that the owner in group code 330 will not be specified. After using DICTADD the owner in group 330 will be the main dictionary.
Right now we have placed a Dictionary named "OUR_DICT" in the main dictionary. To check if it succeeded we can investigate the main dictionary:

Command: (entget (namedobjdict))
((-1 . <Entity name: 16a9860>) (0 . "DICTIONARY") ..etc...
(3 . "ACAD_PLOTSTYLENAME") (350 . <Entity name: 16a9870>)
(3 . "OUR_DICT") (350 . <Entity name: 16a8da0>))

And there it is! But what good does it do us? It just sits there and doesn't hold any data. Well, let's say we want to save data for a routine that creates annotations - for example a text style, a layer name and a text height. Simple stuff, but it'll suffice for the purpose of illustration. With XRecords we can create an entity that can hold any possible data within the range of defined data types. Unlike XData it uses regular group codes to save data.
All group codes (except internal data like code 5 and the negative codes) can be used. Of course, the data types that are defined for the specific group codes have to be respected. This means that, for example, a code 70 cannot hold anything else than a 16-bit integer and so on. Code values can be examined in the DXF Reference.

An XRecord is created in much the same way as a Dictionary. First we'll see if it already exists, then create it without an owner with ENTMAKEX and lastly add it to our custom Dictionary. Again, we will name it in order to retrieve it by name. In this example it will be called "OUR_DICT". Both name and initial values are hardcoded into the function - in the real world we would probably make this a generic function and specify name and values as arguments.

```
(defun get-or-make-Xrecord (/ adict anXrec)

  (cond

    ;;first get our dictionary. Notice that "OUR_DICT" will be
    ;;created here in case it doesn't exist
    ((setq adict (get-or-create-Dict))

     (cond

       ;;if "OUR_DICT" is now valid then look for "OUR_VARS" Xrecord
       ((not (setq anXrec (dictsearch adict "OUR_VARS")))

        ;;if "OUR_VARS" was not found then create it
        (setq anXrec (entmakex '((0 . "XRECORD")
                                 (100 . "AcDbXrecord")
                                 (7 . "Arial")
                                 (8 . "A09--T-")
                                 (40 . 2.0)
                                 )
                       )
        )

        ;;if creation succeeded then add it to our dictionary
        (if anXrec (setq anXrec (dictadd adict "OUR_VARS" anXrec)))
       )

       ;;if it's already present then just return its entity name
       (setq anXrec

        (cdr (assoc -1 (dictsearch adict "OUR_VARS")))
       )
     )
    )
  )
)
```

Now we have an XRecord that contains three different data: a text style name in group code 7, a layer name in group code 8 and a text height in group code 40. All codes are chosen with respect to normal convention, but any code that can be associated with the data type in question can be used. The structure from which to access our data will now be like this:

```
Named object dictionary      =  Dictionary (owner = the drawing)
  > OUR_DICT                 =  Dictionary (owner = named object dictionary)
     > OUR_VARS              =  Xrecord    (owner = OUR_DICT)
         (7 . "Arial")       =    Egenskab i Xrecord
         (8 . "A09-T-")      =    Egenskab i Xrecord
         (40 . 2.0)          =    Egenskab i Xrecord
```

The only thing that remains is to read the data:

```
(defun getvars (/ vars varlist)

  ;;retrieve XRecord "OUR_VARS" from dictionary "OUR_DICT"
  ;;which in turn calls both functions above
  (setq vars (get-or-make-Xrecord))

  ;;if our Xrecord is found, then get values in group code 7, 8 and 40
  (cond (vars
         (setq varlist  (entget vars))
         (setq txtstyle (cdr (assoc 7 varlist)))
         (setq txtlayer (cdr (assoc 8 varlist)))
         (setq txtsize  (cdr (assoc 40 varlist)))
        )

        ;;otherwise return nil
        (T nil)
  )
)
```

Because of the naming scheme, Dictionaries work much like symbol tables in terms of accessing entries. In addition to DICTSEARCH there's also a function, DICTNEXT, to iterate through all entries in a Dictionary. It works like TBLNEXT - here shown by iterating through the main dictionary:

```
(defun C:LISTDICTS (/ maindict adict)

  (setq maindict (namedobjdict))

  (while (setq adict (dictnext maindict (not adict)))

    (princ (cdr (assoc -1 adict)))

    (princ (strcat "\t(type = " (cdr (assoc 0 adict)) ")\n")

    )

  )

  (princ)

)
```

Command: listdicts
<Entity name: 185c0d0> (type = DICTIONARY)
<Entity name: 185f4b8> (type = DICTIONARY)
<Entity name: 185c0d8> (type = DICTIONARY)
<Entity name: 185f4a8> (type = ACDBDICTIONARYWDFLT)
<Entity name: 18c4320> (type = DICTIONARY)
<Entity name: 18c3d10> (type = XRECORD)

There're also functions to rename a Dictionary, DICTRENAME, and to remove a Dictionary,

DICTREMOVE. The latter simply removes its entry from the owner, or in other words: detaches it from the owner. It doesn't delete it unless the owner is "ACAD_GROUP" or "ACAD_MLINESTYLE". Sometimes when updating a Dictionary it's easier to remove/delete it and replace it with a new entry, but that will be for your pleasure to explore.

If you gained some understanding of Dictionaries and XRecords by now then I'll throw in a little assignment: Figure out how you can add the XRecord directly to the main dictionary without first creating a Dictionary!

Drawing Setup

This to me, is the one area that all AutoLispers, especially AutoLisp beginners, should be spending their time and energy.
A good, well written setup routine can saves hours of draughting time, as well as enforcing drawing office standards.

This tutorial will take you step by step through a simple drawing setup routine which, I hope, you will be able to build on to suit your requirements.
We will start by designing a dialogue interface to choose the type of drawing sheet the draughtsman would like to use.

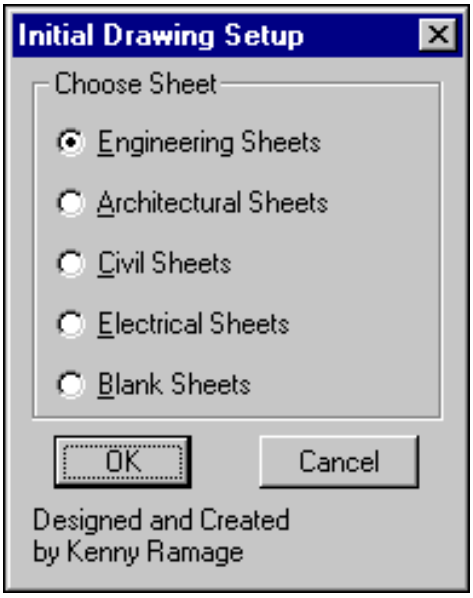
Let's look at the coding for the dialogue box :

```
setup1 : dialog {                                  //dialogue name
        label = "Initial Drawing Setup";           //label
      : boxed_radio_column {                       //start radio column
        label = "Choose Sheet";                    //label
        : radio_button {                           //radio button
          label = "&Engineering Sheets";           //label
          key = "rb1";                             //key
          value = "1";                             //make it default
        }                                          //end radio button
        : radio_button {                           //another radio button
          label = "&Architectural Sheets";         //label
          key = "rb2";                             //key
        }                                          //end radio button
        : radio_button {                           //another radio button
          label = "&Civil Sheets";                 //label
          key = "rb3";                             //key
        }                                          //end radio button
        : radio_button {                           //another radio button
          label = "&Electrical Sheets";            //label
          key = "rb4";                             //key
        }                                          //end radio button
        : radio_button {                           //another radio button
          label = "&Blank Sheets";                 //label
          key = "rb5";                             //key
        }                                          //end radio button
      }                                            //end radio column
        ok_cancel ;                                //OK/Cancel Tile
        : paragraph {                              //begin paragraph
           : text_part {                           //a bit of text
             label = "Designed and Created";       //text
           }                                       //end text
           : text_part {                           //another bit of text
             label = "by Kenny Ramage";            //Credits
           }                                       //end text
        }                                          //end paragraph
      }                                            //end dialogue
```

Now, let's write some lisp to display the dialogue box :

```
(defun C:SETUP1 ()                                  ;define function
     (setvar "BLIPMODE" 0)                          ;switch off variables
     (setvar "CMDECHO" 0)
     (setvar "OSMODE" 0)
     (setq typ "e")                                 ;set default
     (setq dcl_id (load_dialog "setup1.dcl"))       ;load dialogue
     (if (not                                       ;test if loaded
           (new_dialog "setup1" dcl_id)             ;new dialogue
         )                                          ;end not
           (exit)                                   ;if not loaded, exit
     )                                              ;end if
     (set_tile "rb1" "1")                           ;switch on radio button
     (action_tile "rb1"                             ;if button selected
         "(setq typ \"e\")")                        ;store sheet type
     (action_tile "rb2"                             ;if button selected
         "(setq typ \"a\")")                        ;store sheet type
     (action_tile "rb3"                             ;if button selected
         "(setq typ \"c\")")                        ;store sheet type
     (action_tile "rb4"                             ;if button selected
         "(setq typ \"el\")")                       ;store sheet type
     (action_tile "rb5"                             ;if button selected
         "(setq typ \"b\")")                        ;store sheet type
     (action_tile "cancel"                          ;if Cancel selected
         "(done_dialog)(setq userclick nil)")       ;close dialogue, clear flag
     (action_tile "accept"                          ;if OK selected
         "(done_dialog) (setq userclick T)")        ;close dialogue, set flag
     (start_dialog)                                 ;start dialogue
     (unload_dialog dcl_id)                         ;unload dialogue
 (princ)                                            ;finish clean
)                                                   ;end function
(princ)
```

Your dialogue box should look like this :



We've now designed a dialogue box that enables us to choose the type of drawing sheet that we would like to use.
But, we still need to be able to choose the size of sheet and the scale that we would like to draw with.
Let's modify our dialogue box and add these functions :

Please Note : Comments preceded by '*' are new lines.

```
setup2 : dialog {                                   //dialogue name
        label = "Initial Drawing Setup";            //label
      : row {                                       //*start row
      : boxed_radio_column {                        //start radio column
        label = "Choose Sheet";                     //label
        : radio_button {                            //radio button
          label = "&Engineering Sheets";            //label
          key = "rb1";                              //key
          value = "1";                              //make it default
        }                                           //end radio button
        : radio_button {                            //another radio button
          label = "&Architectural Sheets";          //label
          key = "rb2";                              //key
        }                                           //end radio button
        : radio_button {                            //another radio button
          label = "&Civil Sheets";                  //label
          key = "rb3";                              //key
        }                                           //end radio button
        : radio_button {                            //another radio button
          label = "&Electrical Sheets";             //label
          key = "rb4";                              //key
        }                                           //end radio button
        : radio_button {                            //another radio button
          label = "&Blank Sheets";                  //label
          key = "rb5";                              //key
        }                                           //end radio button
      }                                             //end radio column
```

```
    : boxed_radio_column {                          //*start radio column
     label = "Choose Size :";                       //*label
       : radio_button {                             //*radio button
         label = "A&0 - 1189 x 841";                //*label
         key = "rb6";                               //*key
         value = "1";                               //*make it default
       }                                            //*end radio button
       : radio_button {                             //*radio button
         label = "A&1 -   841 x 594";               //*label
         key = "rb7";                               //*key
       }                                            //*end radio button
       : radio_button {                             //*radio button
         label = "A&2 -   594 x 420";               //*label
         key = "rb8";                               //*key
       }                                            //*end radio button
       : radio_button {                             //*radio button
         label = "A&3 -   420 x 297";               //*label
         key = "rb9";                               //*key
       }                                            //*end radio button
       : radio_button {                             //*radio button
         label = "A&4 -   297 x 210";               //*label
         key = "rb10";                              //*key
       }                                            //*end radio button
     }                                              //*end radio column
     }                                              //*end row
       : edit_box {                                 //*start edit box
         label = "&Enter Drawing Scale :" ;         //*label
         key = "eb1" ;                              //*key
         edit_width = 8 ;                           //*width of box
       }                                            //*end edit box
       :spacer { width = 1;}                        //*spacer
       ok_cancel ;                                  //OK/Cancel Tile
       : paragraph {                                //begin paragraph
          : text_part {                             //a bit of text
            label = "Designed and Created";         //text
          }                                         //end text
          : text_part {                             //another bit of text
            label = "by Kenny Ramage";              //Credits
          }                                         //end text
       }                                            //end paragraph
     }                                              //end dialogue
```

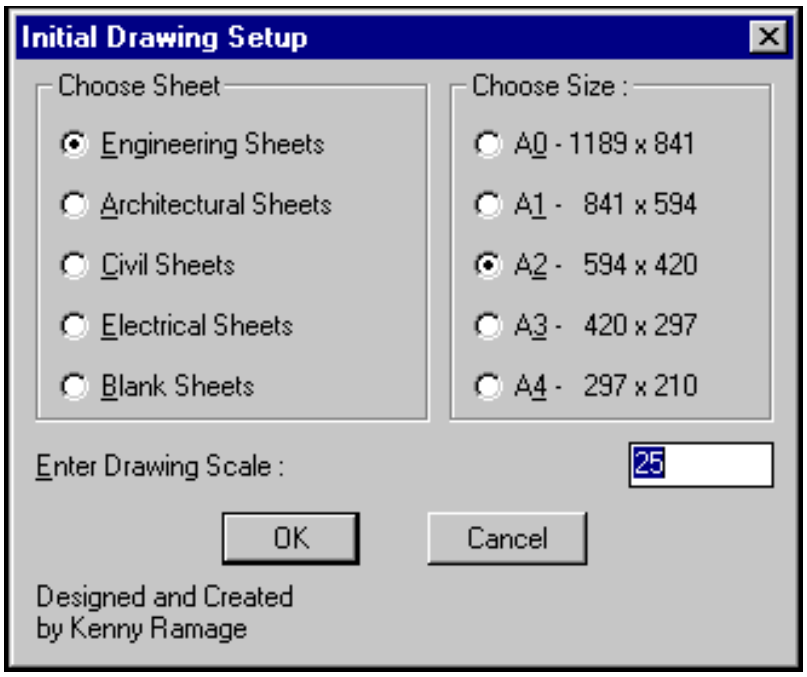
And now the Autolisp Coding :

```
(defun C:SETUP2 ()                                  ;define function
     (setvar "BLIPMODE" 0)                          ;switch off variables
     (setvar "CMDECHO" 0)
     (setvar "OSMODE" 0)
     (setq typ "e")                                 ;set default
     (setq dcl_id (load_dialog "setup2.dcl"))       ;load dialogue
     (if (not                                       ;test if loaded
```

```
      (new_dialog "setup2" dcl_id)          ;new dialogue
    )                                       ;end not
      (exit)                                ;if not loaded, exit
)                                           ;end if
(set_tile "rb1" "1")                        ;switch on radio button
(action_tile "rb1"                          ;if button selected
    "(setq typ \"e\")                       ;store sheet type
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box
(action_tile "rb2"                          ;if button selected
    "(setq typ \"a\")                       ;store sheet type
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box
(action_tile "rb3"                          ;if button selected
    "(setq typ \"c\")                       ;store sheet type
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box
(action_tile "rb4"                          ;if button selected
    "(setq typ \"el\")                      ;store sheet type
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box
(action_tile "rb5"                          ;if button selected
    "(setq typ \"b\")                       ;store sheet type
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box

(set_tile "rb6" "1")                        ;*make default
(set_tile "eb1" "1")                        ;*initial edit box value
(mode_tile "eb1" 2)                         ;*set focus to edit box
(action_tile "rb6"                          ;*if button selected
    "(setq siz \"A0\")                      ;*store sheet size
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box
(action_tile "rb7"                          ;*if button selected
    "(setq siz \"A1\")                      ;*store sheet size
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box
(action_tile "rb8"                          ;*if button selected
    "(setq siz \"A2\")                      ;store sheet size
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box
(action_tile "rb9"                          ;*if button selected
    "(setq siz \"A3\")                      ;*store sheet size
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box
(action_tile "rb10"                         ;*if button selected
    "(setq siz \"A4\")                      ;*store sheet size
     (mode_tile \"eb1\" 2)")                ;*set focus to edit box

(action_tile "cancel"                       ;if Cancel selected
    "(done_dialog)(setq userclick nil)")    ;close dialogue, clear flag
(action_tile "accept"                       ;if OK selected
  (strcat                                   ;*string 'em together
 "(progn (setq #dwgsc                       ;*store value
         (atof (get_tile \"eb1\")))"        ;*get edit box value
       "(done_dialog) (setq userclick T))" ;*close dialogue, set flag
  )                                         ;*end progn
)                                           ;*end action_tile

(start_dialog)                              ;start dialogue
(unload_dialog dcl_id)                      ;unload dialogue
```

```
 (princ)                                          ;finish clean
)                                                 ;end function
(princ)
```

Here's what your dialogue should look like :



We've now completed our dialogue box. Next, we'll try and get it to do something.

As you can see from the dialog box, we have 5 different types of drawing sheets and 5 sizes.
Because of space constraints, I cannot post all 25 template drawings.
Therefore, I will only supply the 5 Blank drawing templates.
All of the template drawings should be named thus :

Engineering Drawings

- EA0.DWG
- EA1.DWG
- EA2.DWG
- EA3.DWG
- EA4.DWG

Architectural Drawings

- AA0.DWG
- AA1.DWG
- AA2.DWG
- AA3.DWG
- AA4.DWG

Civil

- CA0.DWG
- CA1.DWG
- CA2.DWG
- CA3.DWG
- CA4.DWG

Electrical Drawings

- ELA0.DWG
- ELA1.DWG
- ELA2.DWG
- ELA3.DWG
- ELA4.DWG

Blank Drawings

- BA0.DWG
- BA1.DWG
- BA2.DWG
- BA3.DWG
- BA4.DWG

I will leave it up to you to produce the rest of the template drawings.
For your information, here is the dimensions of the most common Metric Drawing Sheets :

```
A0 - 1189 mm x 841 mm
A1 -  841 mm x 594 mm
A2 -  594 mm x 420 mm
A3 -  420 mm x 297 mm
A4 -  297 mm x 210 mm
```

Right, let's get down to writing some more code.
The dialogue box we have designed will return 3 values :

TYP - Type of Sheet (eg. EL for Electrical)
SIZ - Size of Sheet (eg. A3 for A3 size sheet)
#DWGSC - Drawing Scale (eg. 25 for 1 in 25)

We can now use these values to set up our drawing sheet.
Have a look at the following coding :

```
(defun C:SETUP3 ()                                  ;define function
     (setvar "BLIPMODE" 0)                          ;switch off variables
     (setvar "CMDECHO" 0)
     (setvar "OSMODE" 0)
     (setq typ "e")                                 ;set default
     (setq dcl_id (load_dialog "setup3.dcl"))       ;load dialogue
     (if (not                                       ;test if loaded
           (new_dialog "setup3" dcl_id)             ;new dialogue
         )                                          ;end not
           (exit)                                   ;if not loaded, exit
     )                                              ;end if
     (set_tile "rb1" "1")                           ;switch on radio button
     (action_tile "rb1"                             ;if button selected
         "(setq typ \"e\")                          ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb2"                             ;if button selected
         "(setq typ \"a\")                          ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb3"                             ;if button selected
         "(setq typ \"c\")                          ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb4"                             ;if button selected
         "(setq typ \"el\")                         ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb5"                             ;if button selected
         "(setq typ \"b\")                          ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box

     (set_tile "rb6" "1")                           ;*make default
     (setq SIZ "A0")                                ;*Default sheet size
     (set_tile "eb1" "1")                           ;*initial edit box value
     (mode_tile "eb1" 2)                            ;*set focus to edit box
     (action_tile "rb6"                             ;*if button selected
         "(setq siz \"A0\")                         ;*store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb7"                             ;*if button selected
         "(setq siz \"A1\")                         ;*store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb8"                             ;*if button selected
         "(setq siz \"A2\")                         ;store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb9"                             ;*if button selected
         "(setq siz \"A3\")                         ;*store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb10"                            ;*if button selected
         "(setq siz \"A4\")                         ;*store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box

     (action_tile "cancel"                          ;if Cancel selected
         "(done_dialog)(setq userclick nil)")       ;close dialogue, clear flag
     (action_tile "accept"                          ;if OK selected
       (strcat                                      ;*string 'em together
```

```
     "(progn (setq #dwgsc                          ;*store value
             (atof (get_tile \"eb1\")))"           ;*get edit box value
           "(done_dialog) (setq userclick T))" ;*close dialogue, set flag
      )                                          ;*end progn
     )                                           ;*end action_tile

     (start_dialog)                              ;start dialogue
     (unload_dialog dcl_id)                      ;unload dialogue

(if userclick                                    ;*if flag is true
    (progn                                       ;*do the following
           (if (eq siz "A0")                     ;*If size is A0
             (progn                              ;*Do the following
                (setq x 1189.0                   ;*Set x limits
                      y 841.0                    ;*Set y limits
                )                                ;*End setq
             )                                   ;*End progn
            )                                    ;*End If
           (if (eq siz "A1")                     ;*If size is A1
             (progn                              ;*Do the following
                (setq x 841.0                    ;*Set x limits
                      y 594.0                    ;*Set y limits
                )                                ;*End setq
             )                                   ;*End progn
            )                                    ;*End If
           (if (eq siz "A2")                     ;*If size is A2
             (progn                              ;*Do the following
                (setq x 594.0                    ;*Set x limits
                      y 420.0                    ;*Set y limits
                )                                ;*End setq
             )                                   ;*End progn
            )                                    ;*End If
           (if (eq siz "A3")                     ;*If size is A3
             (progn                              ;*Do the following
                (setq x 420.0                    ;*Set x limits
                      y 297.0                    ;*Set y limits
                )                                ;*End setq
             )                                   ;*End progn
            )                                    ;*End If
           (if (eq siz "A4")                     ;*If size is A4
             (progn                              ;*Do the following
                (setq x 297.0                    ;*Set x limits
                      y 210.0                    ;*Set y limits
                )                                ;*End setq
             )                                   ;*End progn
            )                                    ;*End If

           (setq SIZE (strcat TYP SIZ))          ;*Construct name of Template

           (if (eq typ "e") (eng))               ;*Run Engineering Setup
           (if (eq typ "a") (arch))              ;*Run Architectural Setup
           (if (eq typ "c") (civil))             ;*Run Civil Setup
           (if (eq typ "el") (elec))             ;*Run Electrical Setup
           (if (eq typ "b") (blank))             ;*Run Blank Setup

    )                                            ;*End progn
```

```
  )                                                ;*End If

 (princ)                                           ;finish clean
)                                                  ;end function
;;;-----------------------------------------------------------------------
(defun blank ()                                    ;*Define function

      (setvar "DIMSCALE" #DWGSC)                   ;*Set Dimscale
      (setvar "USERR1" #DWGSC)                     ;*Store Scale for future
      (setvar "LTSCALE" (* #DWGSC 10))             ;*Set Ltscale
      (setvar "REGENMODE" 1)                       ;*Regen ON
      (setvar "TILEMODE" 1)                        ;*Tilemode ON
      (setq n (* 3.5 #DWGSC))                      ;*Store Text Height

      (setq L (list (* X #DWGSC) (* Y #DWGSC))     ;*Calculate Limits
      )

      (command "LIMITS" "0,0" L
               "ZOOM" "W" "0,0" L
               "STYLE" "italict"
               "italict" N "" "" "" "" ""
               "INSERT" SIZE "0,0" #DWGSC "" ""
      )                                            ;*Set Up Drawing
       (prompt "\n ")                              ;*Blank Line
      (prompt "\nO.K Setup Routine Complete")      ;*Inform User

 (princ)                                           ;*Exit Quietly
)                                                  ;*End Function
;;;-----------------------------------------------------------------------
(princ)                                            ;*Clean Loading
;;;-----------------------------------------------------------------------
```

First of all, we check that the O.K. button has been selected.
If the user Cancels, nothing happens and we quietly exit the routine.

If O.K. has been selected, we first check the size of sheet chosen then, set our x and y limits depending on what size sheet was chosen.
Next, we take the type and size of sheet and string them together.
This gives us the name of the Template Drawing that we need to insert.
Clever, Hey....

Following this, we test to see what type of sheet was chosen.
Depending on the type, we run a sub-routine to set up our drawing.

I have deliberate allowed different setup routines for each type of drawing. This makes the program fully customisable and flexible.
Please note that these setup routines are set to my standards and may not suit the way you want to setup a drawing.

Did you notice how I stored the scale factor into USERR1?
This allows you to retrieve the scale factor for later use.
(The value in USERR1 is saved with the drawing.)

On the next page I will list the complete coding for all Sheet Types.

Here is the complete AutoLisp Coding :

```
(defun C:SETUP ()                                   ;define function
     (setvar "BLIPMODE" 0)                          ;switch off variables
     (setvar "CMDECHO" 0)
     (setvar "OSMODE" 0)
     (setq typ "e")                                 ;set default
     (setq dcl_id (load_dialog "setup.dcl"))        ;load dialogue
     (if (not                                       ;test if loaded
           (new_dialog "setup" dcl_id)              ;new dialogue
         )                                          ;end not
           (exit)                                   ;if not loaded, exit
     )                                              ;end if
     (set_tile "rb1" "1")                           ;switch on radio button
     (action_tile "rb1"                             ;if button selected
         "(setq typ \"e\")                          ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb2"                             ;if button selected
         "(setq typ \"a\")                          ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb3"                             ;if button selected
         "(setq typ \"c\")                          ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb4"                             ;if button selected
         "(setq typ \"el\")                         ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb5"                             ;if button selected
         "(setq typ \"b\")                          ;store sheet type
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box

     (set_tile "rb6" "1")                           ;*make default
     (setq SIZ "A0")                                ;*Default sheet size
     (set_tile "eb1" "1")                           ;*initial edit box value
     (mode_tile "eb1" 2)                            ;*set focus to edit box
     (action_tile "rb6"                             ;*if button selected
         "(setq siz \"A0\")                         ;*store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb7"                             ;*if button selected
         "(setq siz \"A1\")                         ;*store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb8"                             ;*if button selected
         "(setq siz \"A2\")                         ;store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb9"                             ;*if button selected
         "(setq siz \"A3\")                         ;*store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box
     (action_tile "rb10"                            ;*if button selected
         "(setq siz \"A4\")                         ;*store sheet size
          (mode_tile \"eb1\" 2)")                   ;*set focus to edit box

     (action_tile "cancel"                          ;if Cancel selected
         "(done_dialog)(setq userclick nil)")       ;close dialogue, clear flag
     (action_tile "accept"                          ;if OK selected
       (strcat                                      ;*string 'em together
      "(progn (setq #dwgsc                          ;*store value
```

```
            (atof (get_tile \"eb1\")))"       ;*get edit box value
          "(done_dialog) (setq userclick T))" ;*close dialogue, set flag
      )                                       ;*end progn
    )                                         ;*end action_tile

    (start_dialog)                            ;start dialogue
    (unload_dialog dcl_id)                    ;unload dialogue

(if userclick                                 ;*if flag is true
   (progn                                     ;*do the following
         (if (eq siz "A0")                    ;*If size is A0
           (progn                             ;*Do the following
              (setq x 1189.0                  ;*Set x limits
                    y 841.0                   ;*Set y limits
              )                               ;*End setq
           )                                  ;*End progn
          )                                   ;*End If
         (if (eq siz "A1")                    ;*If size is A1
           (progn                             ;*Do the following
              (setq x 841.0                   ;*Set x limits
                    y 594.0                   ;*Set y limits
              )                               ;*End setq
           )                                  ;*End progn
          )                                   ;*End If
         (if (eq siz "A2")                    ;*If size is A2
           (progn                             ;*Do the following
              (setq x 594.0                   ;*Set x limits
                    y 420.0                   ;*Set y limits
              )                               ;*End setq
           )                                  ;*End progn
          )                                   ;*End If
         (if (eq siz "A3")                    ;*If size is A3
           (progn                             ;*Do the following
              (setq x 420.0                   ;*Set x limits
                    y 297.0                   ;*Set y limits
              )                               ;*End setq
           )                                  ;*End progn
          )                                   ;*End If
         (if (eq siz "A4")                    ;*If size is A4
           (progn                             ;*Do the following
              (setq x 297.0                   ;*Set x limits
                    y 210.0                   ;*Set y limits
              )                               ;*End setq
           )                                  ;*End progn
          )                                   ;*End If

         (setq SIZE (strcat TYP SIZ))         ;*Construct name of Template

         (if (eq typ "e") (eng))              ;*Run Engineering Setup
         (if (eq typ "a") (arch))             ;*Run Architectural Setup
         (if (eq typ "c") (civil))            ;*Run Civil Setup
         (if (eq typ "el") (elec))            ;*Run Electrical Setup
         (if (eq typ "b") (blank))            ;*Run Blank Setup

   )                                          ;*End progn
 )                                            ;*End If
```

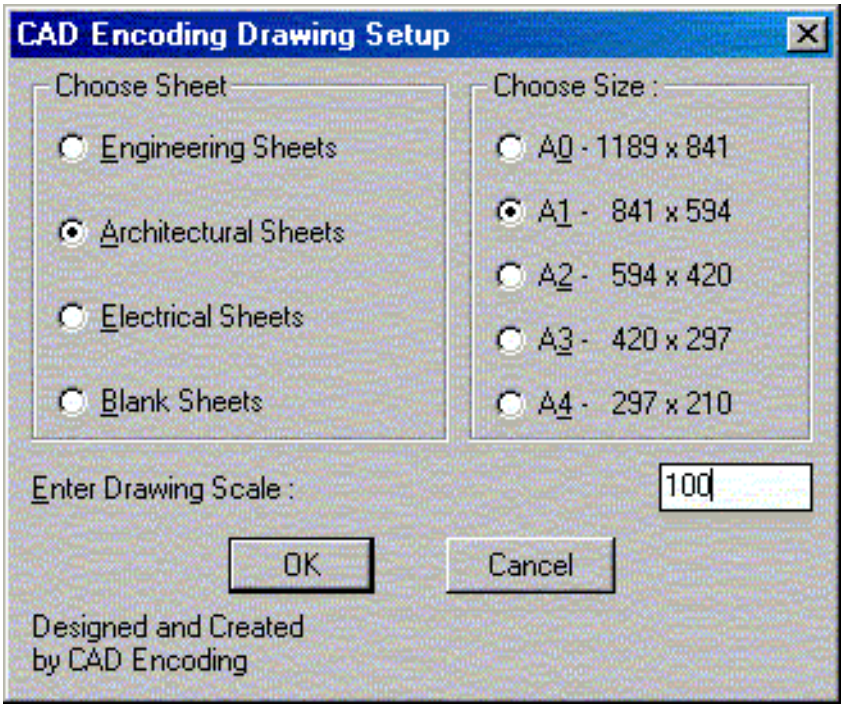
```
 (princ)                                          ;finish clean
)                                                 ;end function
;;;---------------------------------------------------------------------
(defun blank ()                                   ;*Define function

      (setvar "DIMSCALE" #DWGSC)                  ;*Set Dimscale
      (setvar "USERR1" #DWGSC)                    ;*Store Scale for future
      (setvar "LTSCALE" (* #DWGSC 10))            ;*Set Ltscale
      (setvar "REGENMODE" 1)                      ;*Regen ON
      (setvar "TILEMODE" 1)                       ;*Tilemode ON
      (setq n (* 3.5 #DWGSC))                     ;*Store Text Height

      (setq L (list (* X #DWGSC) (* Y #DWGSC))    ;*Calculate Limits
      )

      (command "LIMITS" "0,0" L
               "ZOOM" "W" "0,0" L
               "STYLE" "italict"
               "italict" N "" "" "" "" ""
               "INSERT" SIZE "0,0" #DWGSC "" ""
      )                                           ;*Set Up Drawing
       (prompt "\n ")                             ;*Blank Line
      (prompt "\nO.K Setup Routine Complete")     ;*Inform User

 (princ)                                          ;*Exit Quietly
)                                                 ;*End Function
;;;---------------------------------------------------------------------
(defun eng ()                                     ;*Define function

      (setvar "DIMSCALE" #DWGSC)                  ;*Set Dimscale
      (setvar "USERR1" #DWGSC)                    ;*Store Scale for future
      (setvar "LTSCALE" (* #DWGSC 10))            ;*Set Ltscale
      (setvar "REGENMODE" 1)                      ;*Regen ON
      (setvar "TILEMODE" 1)                       ;*Tilemode ON
      (setq n (* 3.5 #DWGSC))                     ;*Store Text Height

      (setq L (list (* X #DWGSC) (* Y #DWGSC))    ;*Calculate Limits
      )

      (command "LIMITS" "0,0" L
               "ZOOM" "W" "0,0" L
               "STYLE" "italict"
               "italict" N "" "" "" "" ""
               "INSERT" SIZE "0,0" #DWGSC "" ""
      )                                           ;*Set Up Drawing
       (prompt "\n ")                             ;*Blank Line
      (prompt "\nO.K Setup Routine Complete")     ;*Inform User

 (princ)                                          ;*Exit Quietly
)                                                 ;*End Function
;;;---------------------------------------------------------------------
(defun arch ()                                    ;*Define function

      (setvar "DIMSCALE" #DWGSC)                  ;*Set Dimscale
      (setvar "USERR1" #DWGSC)                    ;*Store Scale for future
```

```
      (setvar "LTSCALE" (* #DWGSC 10))          ;*Set Ltscale
      (setvar "REGENMODE" 1)                    ;*Regen ON
      (setvar "TILEMODE" 1)                     ;*Tilemode ON
      (setq n (* 3.5 #DWGSC))                   ;*Store Text Height

      (setq L (list (* X #DWGSC) (* Y #DWGSC))  ;*Calculate Limits
      )

      (command "LIMITS" "0,0" L
               "ZOOM" "W" "0,0" L
               "STYLE" "italict"
               "italict" N "" "" "" "" ""
               "INSERT" SIZE "0,0" #DWGSC "" ""
      )                                         ;*Set Up Drawing
       (prompt "\n ")                           ;*Blank Line
      (prompt "\nO.K Setup Routine Complete")   ;*Inform User

 (princ)                                        ;*Exit Quietly
)                                               ;*End Function
;;;-------------------------------------------------------------------------
(defun civil ()                                 ;*Define function

      (setvar "DIMSCALE" #DWGSC)                ;*Set Dimscale
      (setvar "USERR1" #DWGSC)                  ;*Store Scale for future
      (setvar "LTSCALE" (* #DWGSC 10))          ;*Set Ltscale
      (setvar "REGENMODE" 1)                    ;*Regen ON
      (setvar "TILEMODE" 1)                     ;*Tilemode ON
      (setq n (* 3.5 #DWGSC))                   ;*Store Text Height

      (setq L (list (* X #DWGSC) (* Y #DWGSC))  ;*Calculate Limits
      )

      (command "LIMITS" "0,0" L
               "ZOOM" "W" "0,0" L
               "STYLE" "italict"
               "italict" N "" "" "" "" ""
               "INSERT" SIZE "0,0" #DWGSC "" ""
      )                                         ;*Set Up Drawing
       (prompt "\n ")                           ;*Blank Line
      (prompt "\nO.K Setup Routine Complete")   ;*Inform User

 (princ)                                        ;*Exit Quietly
)                                               ;*End Function
;;;-------------------------------------------------------------------------
(defun elec ()                                  ;*Define function

      (setvar "DIMSCALE" #DWGSC)                ;*Set Dimscale
      (setvar "USERR1" #DWGSC)                  ;*Store Scale for future
      (setvar "LTSCALE" (* #DWGSC 10))          ;*Set Ltscale
      (setvar "REGENMODE" 1)                    ;*Regen ON
      (setvar "TILEMODE" 1)                     ;*Tilemode ON
      (setq n (* 3.5 #DWGSC))                   ;*Store Text Height

      (setq L (list (* X #DWGSC) (* Y #DWGSC))  ;*Calculate Limits
      )
```

```
      (command "LIMITS" "0,0" L
               "ZOOM" "W" "0,0" L
               "STYLE" "italict"
               "italict" N "" "" "" "" ""
               "INSERT" SIZE "0,0" #DWGSC "" ""
      )                                          ;*Set Up Drawing
       (prompt "\n ")                            ;*Blank Line
      (prompt "\nO.K Setup Routine Complete")    ;*Inform User

 (princ)                                         ;*Exit Quietly
)                                                ;*End Function
;;;-------------------------------------------------------------------------


(princ)                                          ;*Clean Loading
;;;-------------------------------------------------------------------------
```

AutoLisp/VBA Drawing Setup

A well written drawing setup routine can save hours in any drawing office and is invaluable when it comes to enforcing drawing office standards. There is nothing worse than a drawing office where everybody "*does there own thing*" in regards to drawing setups. As well as this, a setup routine and template drawings/drawing sheets can be given to third party companies to ensure that they comply with your standards and specifications. This month we're going to have a look at a simple but powerful setup routine first written in plain old AutoLisp and then written using the Dark Side - otherwise known as VBA for the uninitiated. Let's have a look at the program in action :



Usage :

Select your Drawing Sheet type first, then the Drawing Size, select a Scale and then the OK button and away you go.

To use your own drawing sheets in this routine, please refer to the Readme file which is included within the drawing sheet zip file. (The Zip file is included with this tutorial.)

This program not only inserts a chosen type and size of drawing sheet into your drawing, but also sets your text height and dimension scaling to suit the scale factor.
You can even preset your own system variables within the program. Go for it, it's yours for ever and ever to do with as you wish.

First the DCL coding.
Copy and paste this into Notepad and save it as "ALSetup.dcl."

```
//DCL CODING STARTS HERE

alsetup : dialog {
        label = "CAD Encoding Drawing Setup";

      : row {

      : boxed_radio_column {
        label = "Choose Sheet";

        : radio_button {
          label = "&Engineering Sheets";
          key = "rb1";
```

```
        value = "1";
      }
      : radio_button {
        label = "&Architectural Sheets";
        key = "rb2";
      }
      : radio_button {
        label = "&Electrical Sheets";
        key = "rb4";
      }
      : radio_button {
        label = "&Blank Sheets";
        key = "rb5";
      }
    }

   : boxed_radio_column {
    label = "Choose Size :";

      : radio_button {
        label = "A&0 - 1189 x 841";
        key = "rb6";
        value = "1";
      }
      : radio_button {
        label = "A&1 -   841 x 594";
        key = "rb7";
      }
      : radio_button {
        label = "A&2 -   594 x 420";
        key = "rb8";
      }
      : radio_button {
        label = "A&3 -   420 x 297";
        key = "rb9";
      }
      : radio_button {
        label = "A&4 -   297 x 210";
        key = "rb10";
      }
    }
    }

      : edit_box {
        label = "&Enter Drawing Scale :" ;
        key = "eb1" ;
        edit_width = 8 ;
      }
```

```
        :spacer { width = 1;}

        ok_cancel ;

        : paragraph {

           : text_part {
             label = "Designed and Created";
           }

           : text_part {
             label = "by CAD Encoding";
           }

        }

      }

//DCL CODING ENDS HERE
```

And next, here's the AutoLisp coding. No Visual Lisp this time, just plain AutoLisp. ("Phew," I can hear a lot of you saying, wiping your delicate brows in relief.)
Copy and paste this into Notepad and save it as "ALSetup.lsp."

```
;CODING STARTS HERE

(prompt "\nCAD Encoding Setup loaded - Type \"ALSETUP\" to run....")

(defun C:ALSETUP ( / userclick dcl_id siz typ x y l size n oldblip
                                    oldecho oldsnap #dwgsc)

    ;save system settings
    (setq oldblip (getvar "BLIPMODE")
          oldecho (getvar "CMDECHO")
          oldsnap (getvar "OSMODE")
    );setq

    ;set system variables
    (setvar "BLIPMODE" 0)
    (setvar "CMDECHO" 0)
    (setvar "OSMODE" 0)

    ;load the dialog
    (setq dcl_id (load_dialog "alsetup.dcl"))
    (if (not (new_dialog "alsetup" dcl_id))
        (exit)
    );if
```

```
    ;define default settings
    (set_tile "rb1" "1")
    (set_tile "rb6" "1")
    (set_tile "eb1" "1")
    (mtile)
    (setq siz "A0")
    (setq typ "e")


    ;define radio buttons action statements
    (action_tile "rb1" "(setq typ \"e\")  (mtile)")
    (action_tile "rb2" "(setq typ \"a\")  (mtile)")
    (action_tile "rb4" "(setq typ \"el\") (mtile)")
    (action_tile "rb5" "(setq typ \"b\")  (mtile)")

    (action_tile "rb6" "(setq siz \"A0\") (mtile)")
    (action_tile "rb7" "(setq siz \"A1\") (mtile)")
    (action_tile "rb8" "(setq siz \"A2\") (mtile)")
    (action_tile "rb9" "(setq siz \"A3\") (mtile)")
    (action_tile "rb10" "(setq siz \"A4\") (mtile)")

    ;define Cancel action statements
    (action_tile "cancel"
        "(done_dialog)(setq userclick nil)")

    ;define OK action statements
    (action_tile "accept"
      (strcat
     "(progn (setq #dwgsc (atof (get_tile \"eb1\")))"
          "(done_dialog) (setq userclick T))"
      );progn
    )

    ;display the dialog
    (start_dialog)

    ;unload the dialog
    (unload_dialog dcl_id)

;check the flag setting
(if userclick

    ;if it's set, do the following
    (progn

    ;set up the sheet size
    (cond

      ((= siz "A0") (setq x 1189.0 y 841.0))
      ((= siz "A1") (setq x 841.0 y 594.0))
```

```
      ((= siz "A2") (setq x 594.0 y 420.0))
      ((= siz "A3") (setq x 420.0 y 297.0))
      ((= siz "A4") (setq x 297.0 y 210.0))

    );cond

    ;Construct drawing sheet name
     (setq size (strcat typ siz))

      ;set system variables according to scale
      (setvar "DIMSCALE" #dwgsc)
      (setvar "USERR1" #dwgsc)
      (setvar "LTSCALE" (* #dwgsc 10))
      (setvar "REGENMODE" 1)
      (setvar "TILEMODE" 1)
      (setq n (* 3.5 #dwgsc))

      ;define the limits list
      (setq L (list (* x #dwgsc) (* y #dwgsc)))

      ;set up the drawing
      (command "LIMITS" "0,0" L
                       "ZOOM" "W" "0,0" L
                   "STYLE" "italict" "italict" N "" "" "" "" ""
                       "INSERT" size "0,0" #dwgsc "" "" )

        ;inform the user
        (prompt "\n ")
        (prompt "\nOkay - Setup Routine Complete")

    );progn

  );if

  ;reset system variables
  (setvar "BLIPMODE" oldblip)
  (setvar "CMDECHO" oldecho)
  (setvar "OSMODE" oldsnap)

 ;finish clean
 (princ)

);defun
;;;---------------------------

;function to set the focus to
;the scale edit box
(defun mtile ()

(mode_tile "eb1" 2)
```

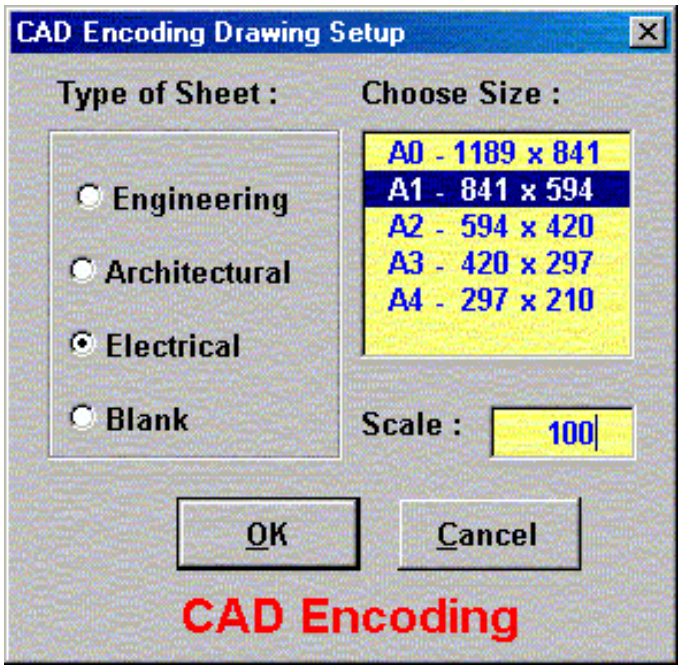
```
);defun
;;;-------------------------

;load clean
(princ)

;CODING ENDS HERE
```

Store both files and the drawing sheet files in a directory within your AutoCAD search path. Now type *(load "ALSetup")* at the AutoCAD command prompt, and then *"ALSetup"* to run the setup program.

On the next page we'll have a look at doing the same thing using VBA, but this time we'll make use of drawing template files.

Welcome to the Dark Side!!

Just to be different, instead of using drawing files, this time we are going to use drawing template files within our setup routine.
As well as that, just to show off, we are going to make our dialog a wee bit more colorful than the boring old AutoLisp dialog. (eat your hearts out Lispers.)
Okay, here's a preview of what our dialog will look like on completion :



Usage :

Select your Drawing Sheet type first, then the Drawing Size, select a Scale and then the OK button and away you go.

To use your own template drawings in this routine, please refer to the Readme file which is included within the drawing sheet zip file.

As for the AutoLisp version, you can customise this program to your little hearts desire.

Right, let's have a look at a wee bit of coding. Fire up AutoCAD and open a new Project in the Visual Basic Editor.
First you need to insert a new UserForm keeping the default name. (UserForm1)
Now add the following controls, naming them as shown :

- **Button - "*cmdOk*"**
- **Button - "*cmdCancel*"**
- **Frame containing 4 Option Buttons - "*Opt1*", "*Opt2*", "*Opt3*" and "*Opt4*".**
- **Listbox - "*Listbox1*"**
- **Edit Box - "*Textbox1*"**

Now add the following coding under General Declarations :

```
Option Explicit

'------------------------------------------------

Private Sub cmdCancel_Click()
End
End Sub

'------------------------------------------------

Private Sub cmdOk_Click()

'declare variables
Dim acadApp As Object
```

```
Dim acadDoc As Object
Dim doc As Object
Dim templateFileName As String
Dim DrgSheet As String
Dim DrgSize As String
Dim newLimits(0 To 3) As Double
Dim sc As Double
Dim oldTextStyle As Object
Dim Blockref As Object
Dim InsertionPoint(0 To 2) As Double
Dim Mspace As Object
Dim Insertsheet As String

'hide the dialogue box
Me.Hide

'set a reference to the AutoCAD Application
Set acadApp = GetObject(, "AutoCAD.Application")

    'if there is an error
    If Err Then

        'inform user
        MsgBox Err.Description

        'exit application
        Exit Sub

    End If

'set reference to active document
Set acadDoc = acadApp.ActiveDocument

'if the current drawing is not saved
If Not acadDoc.Saved Then

    'ask the user what he wants to do
    If MsgBox("    OK to Save Drawing?", 4) = vbNo Then

    'if No end application
    End

    Else

        'if Yes save the drawing
        acadDoc.Save

    End If

End If

'get the scale and convert to double
sc = CDbl(TextBox1.Text)

'if this button selected
If Opt1.Value = True Then
```

```
    'set first letter of drawing sheet
    DrgSheet = "E"

    'set the relevant template file
    templateFileName = "acadeng.dwt"

    'open new drawing with selected template file
    Set doc = acadDoc.New(templateFileName)

End If

If Opt2.Value = True Then
    DrgSheet = "A"
    templateFileName = "acadarch.dwt"
    Set doc = acadDoc.New(templateFileName)
End If

If Opt3.Value = True Then
    DrgSheet = "EL"
    templateFileName = "acadelec.dwt"
    Set doc = acadDoc.New(templateFileName)
End If

If Opt4.Value = True Then
    DrgSheet = "B"
    templateFileName = "acadblank.dwt"
    Set doc = acadDoc.New(templateFileName)
End If

'get the item selected from list box
Select Case ListBox1.ListIndex

'if the index is 0 (first item)
Case 0

    'set the drawing size
    DrgSize = "A0"

    'set the limits
    newLimits(0) = 0#
    newLimits(1) = 0#
    newLimits(2) = 1189# * sc
    newLimits(3) = 841# * sc

    'get the current text style
    Set oldTextStyle = acadDoc.ActiveTextStyle

Case 1
    DrgSize = "A1"
    newLimits(0) = 0#
    newLimits(1) = 0#
    newLimits(2) = 841# * sc
    newLimits(3) = 594# * sc
    Set oldTextStyle = acadDoc.ActiveTextStyle
Case 2
    DrgSize = "A2"
```

```
    newLimits(0) = 0#
    newLimits(1) = 0#
    newLimits(2) = 594# * sc
    newLimits(3) = 420# * sc
    Set oldTextStyle = acadDoc.ActiveTextStyle
Case 3
    DrgSize = "A3"
    newLimits(0) = 0#
    newLimits(1) = 0#
    newLimits(2) = 420# * sc
    newLimits(3) = 297# * sc
    Set oldTextStyle = acadDoc.ActiveTextStyle
Case 4
    DrgSize = "A4"
    newLimits(0) = 0#
    newLimits(1) = 0#
    newLimits(2) = 297# * sc
    newLimits(3) = 210# * sc
    Set oldTextStyle = acadDoc.ActiveTextStyle
End Select

'set drawing limits
acadDoc.Limits = newLimits

'zoom to extents
ZoomExtents

'set Ltscale
Call acadDoc.SetVariable("Ltscale", sc * 10)

'set Dimscale
Call acadDoc.SetVariable("Dimscale", sc)

'store scale in Userr1 for later use
Call acadDoc.SetVariable("Userr1", sc)

'set Regenmode
Call acadDoc.SetVariable("Regenmode", 1)

'set Tilemode
Call acadDoc.SetVariable("Tilemode", 1)

'set Text Height
oldTextStyle.Height = 3.5 * sc

'String Drawing Sheet Name Together
Insertsheet = DrgSheet & DrgSize

'set the insertion point
InsertionPoint(0) = 0
InsertionPoint(1) = 0
InsertionPoint(2) = 0

'get reference to Model Space
Set Mspace = acadDoc.ModelSpace
```

```
'Insert the drawing sheet
Set Blockref = Mspace.InsertBlock(InsertionPoint, Insertsheet, sc, sc, sc, 0)

End

End Sub

'-------------------------------------------------

Private Sub ListBox1_Click()
TextBox1.SetFocus
End Sub

'-------------------------------------------------

Private Sub ListBox1_DblClick(ByVal Cancel As MSForms.ReturnBoolean)
cmdOk_Click
End Sub

'-------------------------------------------------

Private Sub Opt1_Click()
TextBox1.SetFocus
End Sub

'-------------------------------------------------

Private Sub Opt2_Click()
TextBox1.SetFocus
End Sub

'-------------------------------------------------

Private Sub Opt3_Click()
TextBox1.SetFocus
End Sub

'-------------------------------------------------

Private Sub Opt4_Click()
TextBox1.SetFocus
End Sub

'-------------------------------------------------

Private Sub UserForm_Initialize()

'Populate the List Box
ListBox1.AddItem "A0 - 1189 x 841"
ListBox1.AddItem "A1 -  841 x 594"
ListBox1.AddItem "A2 -  594 x 420"
ListBox1.AddItem "A3 -  420 x 297"
ListBox1.AddItem "A4 -  297 x 210"
ListBox1.ListIndex = 0
```

```
End Sub
```

Next, you need to add a new Module.
Then add this coding :

```
Sub VbaSetup()
UserForm1.Show
End Sub
```

Save your project as "*VbaSetup.dvb*", then run the macro "*VbaSetup*."
The dialog should appear in all it's glory. Congratulations!
Quick, call your colleagues, call your boss, in fact, call all your friends and family to come and have a look and what you've done. Who's a clever boy then? (sorry, or girl.)

Application Data.

Have you ever looked at the ACAD2002.CFG file? (Autocad Configuration File)
This is simply a text file divided into sections.
The only section that you have control over is the [AppData] section.
Let's do something with it.

Say you wanted to store 2 bits of information that you use in one of your AutoLisp routines. You would store them like this :

```
(setcfg "appdata/test/bolt" "M24")
(setcfg "appdata/test/length "90")
```

After (setcfg must come "appdata" followed by your application name, which must be unique. Following this is the variable tag and then the information you wish to store. If you open the ACAD2002.CFG file, this is what the [AppData] section will look like :

```
[AppData/test]
bolt=4
length=90
```

To retrieve this data you would use the following syntax :

```
(setq a (getcfg "appdata/test/bolt"))
(setq b (getcfg "appdata/test/length"))
```

This would set a to "M24" and b to "90".
Now this is great, I hear you say, but were would you use it.
Here's a good example for you :

Say you've written a routine that you would like other people to use on a trial basis, say 5 times. After the 5 times they either pay up, or the routine ceases to work.

What we are going to do is first write a simple routine.
Then we will write a sub-routine that stores the number of times the routine is run in the [AppData] section. Next, we will check the number of times it has been run and then decide whether the limit has expired or not. Have a look :

```
;Main function

(defun c:test8 ()
        (register)
        (if flagset
                (alert "\nThe programme will now run.")
        );if
   (princ)
)defun
```

```
;Sub-function

(defun register ()
        (setq v1 (getcfg "AppData/CadKen/test"))
           (if (= v1 nil)
             (progn
                 (setcfg "AppData/CadKen/test" "1")
                 (setq v1 (getcfg "AppData/CadKen/test"))
             );progn
           );if
        (setq v2 5)
        (setq v1 (atoi v1))
           (if (< v1 v2)
             (progn
                (setq v1 (+ v1 1))
                  (cond
                        ((= v1 1)  (setcfg "AppData/CadKen/test" "1"))
                        ((= v1 2)  (setcfg "AppData/CadKen/test" "2"))
                        ((= v1 3)  (setcfg "AppData/CadKen/test" "3"))
                        ((= v1 4)  (setcfg "AppData/CadKen/test" "4"))
                        ((= v1 5)  (setcfg "AppData/CadKen/test" "5"))
                  );cond
                  (setq v3 v1)
                  (setq v3 (rtos (- v2 v1)))
                  (alert (strcat "\n You Have " v3 " more Loadings Left"))
                  (setq flagset T)
             );progn
             (progn
                (alert "\nEvalution Period has Expired
                        \n   Contact Kenny Ramage
                        \n      ndbe51d1@db.za
                        \n     For Licensed Copy")
                (setq flagset nil)
             );progn
        );if
    (princ)
);defun
(princ)
```

This is a very simple but usefull routine that demonstrates the use of [AppData]. You would still, of course, have to encrypt or compile your routine to stop the user simply changing the variable that contains the usage control number.

Date and Time Stamping.

When you plot a drawing (or insert an Xref, etc), it's nice to have the Date and Time of the plot stamped on the plotted drawing. This is easily done manually, but wouldn't it be nice to use AutoLisp and have the Time and Date automatically added to your plots? Here's how you would go about it.

First, we need to write a few functions that will calculate the Date and Time and then format them into readable text. Try out the following functions :

```
(defun TODAY ( / d yr mo day)
;define the function and declare all variabled local

     (setq d (rtos (getvar "CDATE") 2 6)
     ;get the date and time and convert to text

          yr (substr d 3 2)
          ;extract the year

          mo (substr d 5 2)
          ;extract the month

         day (substr d 7 2)
         ;extract the day

     );setq

     (strcat day "/" mo "/" yr)
     ;string 'em together

  (princ)

);defun
;;;*----------------------------------------
(defun TIME ( / d hr m s)
;define the function and declare all variables as local

     (setq d (rtos (getvar "CDATE") 2 6)
     ;get the date and time and convert to text

          hr (substr d 10 2)
          ;extract the hour

          m (substr d 12 2)
          ;extract the minute

          s (substr d 14 2)
          ;extract the second

     );setq
```

```
    (strcat hr ":" m ":" s)
    ;string 'em together

  (princ)

);defun
;;;*----------------------------------------
```

Try the two functions out. Load the functions and type:

```
        (today)
```

Lisp should return today's date :

```
        "23/03/99"
```

Now try the next function. Type :

```
        (time)
```

AutoLisp should return something like :

```
        "11:36:21"
```

O.K. great, we've got the date and time but now we need to add it to our drawing. The simplest way of doing this is by making use of attributes.

First we need to create a block containing 3 attributes, namely Date, Time and Who By. (If you don't know how to create a block with attributes, please refer to the AutoCad Reference Manual.)

Once you have created your attribute block, it is very easy to write an AutoLisp routine that inserts the block and automatically calculates and fills in the attribute data. The following example does just that.

```
(defun C:TIMESTAMP (/ ss1 count emax en ed blkn found
                      thedate thetime plotby)
;define function and declare variables as local

        (setvar "HIGHLIGHT" 0)
        ;switch off highlight

        (setvar "CMDECHO" 0)
        ;switch off command echo

        (setq ss1 (ssget "X" '((0 . "INSERT")(66 . 1))))
        ;filter for all blocks with attributes
```

```
(if ss1
;if any are found

  (progn
  ;do the following

        (setq count 0
        ;set the counter to zero

        emax  (sslength ss1)
        ;get the number of blocks

        );setq

        (while (< count emax)
        ;while the counter is less than the
        ;number of blocks

                (setq en   (ssname ss1 count)
                ;get the entity name

                ed (entget en)
                ;get the entity list

                blkn (dxf 2 ed)
                ;get the block name

                );setq

                (if (= "STAMP")
                ;if the block name is "STAMP"

                        (setq   count emax
                        ;stop the loop

                        found T
                        ;set the flag

                        );setq

                        (setq count (1+ count))
                        ;if not increment the counter

                );end if

        );while & if

        (if found
        ;if the flag is set

        (command "ERASE" en "")
```

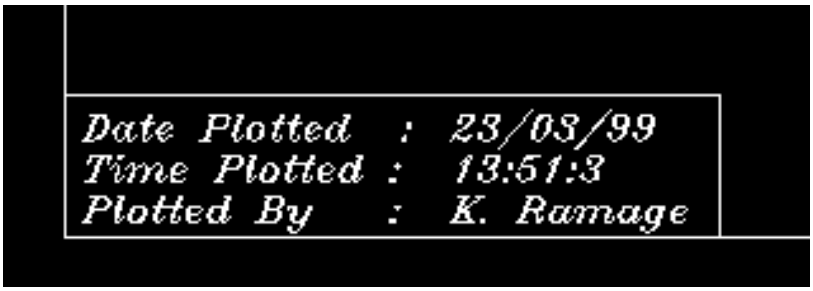
```
                ;erase the block

                );if

            );progn

        );if

        (setvar "ATTDIA" 0)
        ;switch off dialogue boxes

        (setq thedate (today))
        ;calculate and format date

        (setq thetime (time))
        ;calculate and format time

        (setq plotby (getvar "LOGINNAME"))
        ;get the users name

        (command "Insert" "Stamp" "0,0" "" "" ""
                 thedate thetime plotby)
        ;insert the block and fill in the attribute data

        (setvar "ATTDIA" 1)
        ;switch the dialogues back on

        (setvar "HIGHLIGHT" 1)
        ;switch Highlight On

        (setvar "CMDECHO" 1)
        ;switch Cmdecho On

  (princ)

);defun

;=============================================================
(defun dxf(code elist)

  (cdr (assoc code elist))
  ;finds the association pair, strips 1st element

);defun
;=============================================================
(defun TODAY ( / d yr mo day)
     (setq d (rtos (getvar "CDATE") 2 6)
          yr (substr d 3 2)
          mo (substr d 5 2)
         day (substr d 7 2)
     );setq
```

```
     (strcat day "/" mo "/" yr)
);defun
;;;*-------------------------------------------------------------
(defun TIME ( / d hr m s)
     (setq d (rtos (getvar "CDATE") 2 6)
          hr (substr d 10 2)
           m (substr d 12 2)
           s (substr d 14 2)
     );setq
     (strcat hr ":" m ":" s)
);defun
;;;*-------------------------------------------------------------
(princ)
```

This routine basically does the following :

First of all, it checks if there are any blocks in the drawing that contain attributes. If there are, it loops through each block checking if it has the name "STAMP". If it has, it deletes the block. The routine then inserts a new "STAMP" block with the updated attribute data.

After running this routine, you should have a block similar to this in the lower left hand corner of your drawing.



You can now plot your drawing with the new or updated Date and Time stamp.

You should find it relatively easy to modify this routine to suit any kind of Date/Time stamp that you would wish to add to a drawing. You could add a stamp that places the name and path of all Xrefs attached to the drawing, along with, the date they where added and who added them.

Macro Recorder.

Another useful aspect to programs such as AutoLISP is their ability to perform repetitive tasks. For example, suppose you want to be able to record a series of keyboard entries as a macro. One way to do this would be to use a series of Getstring functions as in the following:

```
(Setq str1 (getstring "\nEnter macro: "))

(Setq str2 (getstring "\nEnter macro: "))

(Setq str3 (getstring "\nEnter macro: "))

(Setq str4 (getstring "\nEnter macro: "))
```

Each of the str variables would then be combined to form a variable storing the keystrokes. Unfortunately, this method is inflexible. It requires that the user input a fixed number of entries, no more and no less. Also, this method would use memory storing each keyboard entry as a single variable.

It would be better if you had some facility to continually read keyboard input regardless of how few or how many different keyboard entries are supplied. The While function can be used to repeat a prompt and obtain data from the user until some test condition is met. Here is the coding for our Macro Recorder:

```
(defun C:MACRO (/ str1 macro macname)

        (setq macro '(command))
        ;start list with command

        (setq macname (getstring "\nEnter name of macro: "))
        ;get name of macro

        (while (/= str1 "/")
        ;do while str1 not equal to /

                (setq str1 (getstring "\nEnter macro or / to exit: " ))
                ;get keystrokes

                (if (= str1 "/")

                        (princ "\nEnd of macro ")
                        ;if / then print message

                        (Setq macro (append macro (list str1)))
                        ;else append keystrokes to list
                )
                ;end if macro list
```

```
      );end while

       (eval (list 'defun (read macname) '() macro))
       ;create function

   (princ)

);end macro

(princ)
```

Now we will use the macro program to create a keyboard macro that changes the last object drawn to layer 3. Do the following:

1. Draw a diagonal line from the lower left corner of the drawing area to the upper right corner.

2. Load the Macro.lsp file

3. Enter Macro at the command prompt.

4. At the following prompt:

   Enter name of macro:

5. Enter the word 'CHLA'.

6. At the prompt:

   Enter macro or / to exit:

  Enter the word

   CHANGE

The Enter macro prompt appears again. Enter the following series of words at each Enter macro prompt:

Enter macro or / to exit: L

Enter macro or / to exit: press return

Enter macro or / to exit: P

Enter macro or / to exit: LA

Enter macro or / to exit: 3

Enter macro or / to exit: press return

```
Enter macro or / to exit: press return

Enter macro or / to exit: /
```

**This is where the 'while' function takes action. As you enter each response to the Enter macro prompt, 'while' tests to see if you entered a forward slash.
If not, it evaluates the expressions included as its arguments. Once you enter the backslash, 'while' stops its repetition. You get the prompt:**

```
End of macro
```

The input you gave to the Enter macro prompt is exactly what you would enter if you had used the change command normally. Now run your macro by entering:

```
(chla)
```

The line you drew should change to layer 3.

When Macro starts, it first defines two variables def and macro.

```
(setq def "defun ")

(setq macro '(command))
```

Def is a string variable that is used later to define the macro. Macro is the beginning of a list variable which is used to store the keystrokes of the macro. The next line prompts the user to enter a name for the macro.

```
(setq macname (getstring "\nEnter name of macro: "))
```

**The entered name is then stored with the variable macname.
Finally, we come to the 'while' function.**

```
(while (/= str1 "/")
```

**The 'while' expression is broken into several lines. The first line contains the actual while function along with the test expression.
In this case, the test compares the variable str1 with the string "/" to see if they are not equal. So long as str1 is not equal to "/", 'while' will execute the arguments that follow the test. The next four lines are the expressions to be evaluated. The first of these lines prompts the user to enter the text that compose the macro.**

```
(setq str1 (getstring "\nEnter macro or / to exit: " ))
```

When the user enters a value at this prompt, it is assigned to the variable str1.

The next line uses an if function to test if str1 is equal to "/".

```
(if (= str1 "/")
```

If the test results in T, the next line prints the string :

```
End of macro

(princ "\nEnd of macro ")
```

This expression prints the prompt End of macro to the prompt line.
If the test results in nil, the following line appends the value of str1 to the existing list macro.

```
(Setq macro (append macro (list str1)))
```

The 'append' function takes one list and appends its contents to the end of another list. In this case, whatever is entered at the Enter macro prompt is appended to the variable macro. Each time a new value is entered at the Enter macro prompt, it is appended to the variable macro creating a list of keyboard entries to be saved.

The next two lines close the if and while expressions.

```
);end if

);end while
```

The last line combines all the elements of the program into an expression that, when evaluated, creates a new macro program.

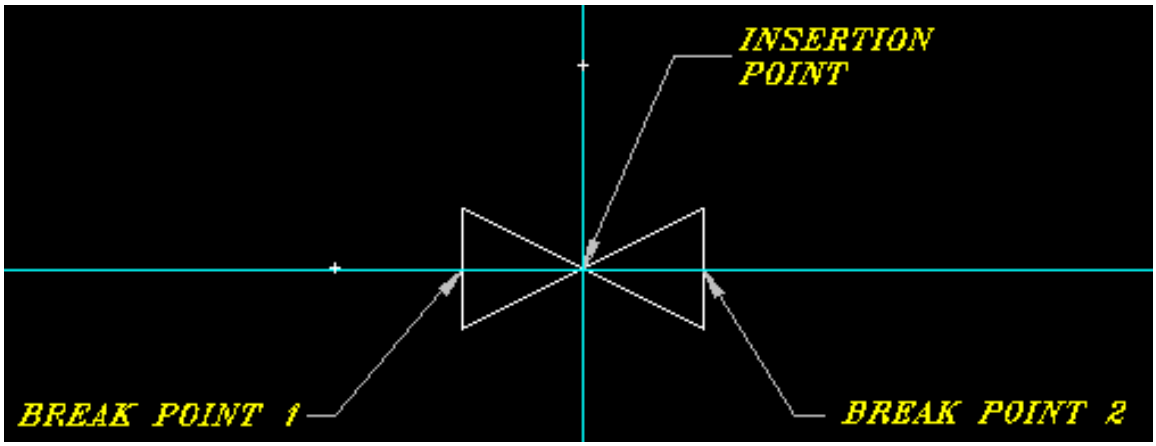
```
(eval (list (read def) (read macname) '() macro))
```

The read function used in this expression is a special function that converts a string value into a symbol. If a string argument to read contains spaces, read will convert the first part of the string and ignore everything after the space.

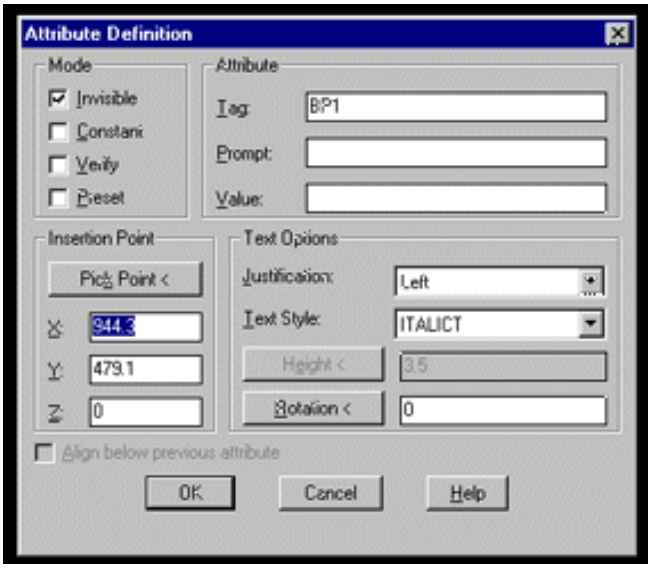
Auto-Breaking Blocks

**I've had a lot of request to write a tutorial on creating Auto-Breaking blocks.
These are very handy little objects and can save a draughtsman a lot of time and effort. The only drawback to them is that they initially take a bit of time to prepare. Let's have a look at what we need to do to go about creating Auto-Breaking
blocks.**

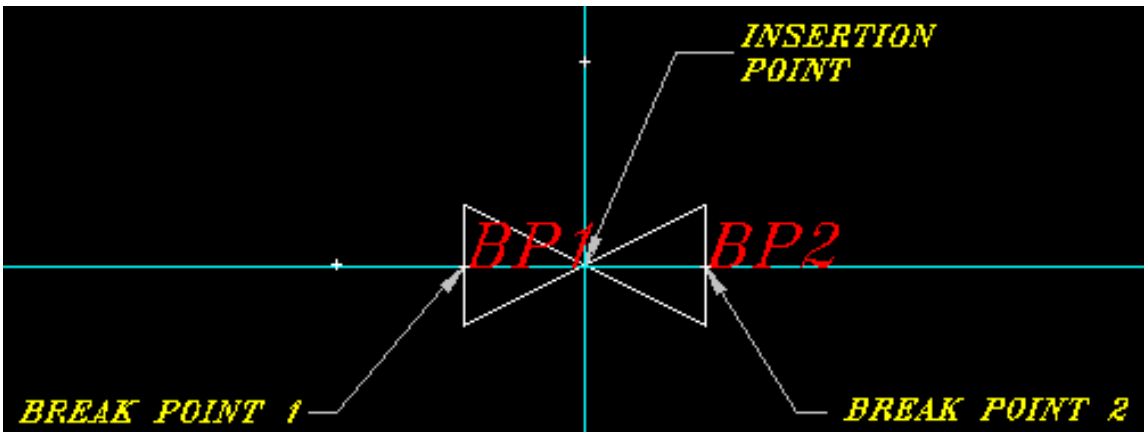
**Firstly, we need to find some way of "storing" the break point data within the block.
For this tutorial we are going to use attributes. Fire up AutoCAD and draw a valve similar to the one shown here.**



Now type in "DDATTDEF" at the command prompt. The Attribute definition dialogue box will appear.



**In the mode section, switch on the "Invisible" toggle, name the Attribute Tag "BP1" and select Break Point 1 as the insertion point. Then select O.K.
Repeat the process for Break Point 2 and your valve is ready to be saved as a Wblock. Your valve should look something like this :**



Now WBlock the valve to your hardrive.

O.K. Now we've stored the Break Point values as attributes within our block.
Now comes the difficult part!! How do we retrieve this information? Easy....
First, open a new drawing and insert the Valve drawing. Just select "O.K." when the attribute dialog appears. Now type this at the command line:

```
(setq edata (entget (setq en (entlast))))
```

Lisp should return something like this :

```
((-1 . ) (0 . "INSERT") (5 . "7AE") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (100 . "AcDbBlockReference") (66 . 1) (2 . "VALVE") (10
508.26 476.045 0.0) (41 . 1.0) (42 . 1.0) (43 . 1.0) (50 . 0.0) (70 . 0) (71 .0)
(44 . 0.0) (45 . 0.0) (210 0.0 0.0 1.0))
```

O.K. we've got the entity data list, but now we need the attribute data. Type this:

```
(setq edata (entget (entnext (cdr (assoc -1 edata)))))
```

You should get something like this:

```
((-1 . ) (0 . "ATTRIB") (5 . "7AF") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (100 . "AcDbText") (10 518.26 476.045 0.0) (40 . 3.5) (1 ."")
(50 . 0.0) (41 . 1.0) (51 . 0.0872665) (7 . "ITALICT") (71 . 0) (72 . 0)
(11 0.0 0.0 0.0) (210 0.0 0.0 1.0) (100 . "AcDbAttribute") (2 . "IP2") (70 . 1)
(73 . 0) (74 . 0))
```

Voila, the attribute data list. Now, I just happen to know that the DXF group code for the attribute insertion point is Code 10. So let's extract it :

```
(setq ip1 (cdr (assoc 10 edata)))
```

Lisp should return the attribute insertion point and it should look something like this:

```
(518.26 476.045 0.0)
```

We would now simply repeat the process for the second break point.

Well that, is basically the heart of the program. We've inserted our block and we've established our break points. All we need to do now is break our line.

Here's the full AutoLISP coding for our Auto-Breaking Block:

```
(defun c:abreak ( / oldsnap bname ip ent1 ent2 ep1 ep2 ang edata ip1 ip2)

        (setq oldsnap (getvar "OSMODE"))
        ;get the current snap

        (setvar "OSMODE" 544)
        ;set snap to intersection and nearest

        (setvar "BLIPMODE" 0)
        ;switch blips off

        (setvar "CMDECHO" 0)
        ;switch command echo off

        (setq bname (getfiled "Select Auto-Breaking Block" "" "dwg" 8))
        ;get the block to insert

        (while
        ;while an insertion point is selected

        (setq ip (getpoint "\nInsertion Point: "))
        ;get the insertion point

        (setq ent1 (entsel "\nSelect Line to AutoBreak: "))
        ;get the line to break

        (setvar "OSMODE" 0)
        ;switch the snap off

        (setq ent2 (entget (car ent1)))
        ;get the entity data of the line

        (setq ep1 (cdr (assoc 10 ent2)))
        ;get the first end point

        (setq ep2 (cdr (assoc 11 ent2)))
        ;get the second end point

        (setq ang (angle ep1 ep2))
        ;get the angle of the line

        (setq ang (/ (* ang 180.0) pi))
        ;convert it to degrees

        (setvar "ATTDIA" 0)
        ;switch off the attribute dialog box

        (command "Insert" bname ip "" "" ang "" "")
        ;insert the block

        (setq edata (entget (setq en (entlast))))
        ;get the block entity data

        (setq edata (entget (entnext (dxf -1 edata))))
        ;get the attribute entity list
```

```
        (setq ip1 (dxf 10 edata))
        ;extract the first attribute insertion point

        (setq edata (entget (entnext (dxf -1 edata))))
        ;get the next attribute entity list

        (setq ip2 (dxf 10 edata))
        ;extract the second attribute insertion point

        (command "Break" ent1 "f" ip1 ip2)
        ;break the line

        (setvar "OSMODE" 544)
        ;switch snap back on

        );while

        (setvar "OSMODE" oldsnap)
        ;reset snap

        (setvar "BLIPMODE" 1)
        ;switch blips back on

        (setvar "CMDECHO" 1)
        ;switch command echo back on

        (setvar "ATTDIA" 1)
        ;switch attribute dialog boc back on

(princ)
;finish clean

);defun

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

(defun dxf (code elist)

  (cdr (assoc code elist))

);defun

(princ)
```

I have created a special folder called "Autobreak" were I store all my Auto-Breaking blocks. I also include this folder as my default directory in the 'getfiled' function.

If you would like the source coding for this application plus the sample Valve drawing, then just click here.

List Variables/Functions.

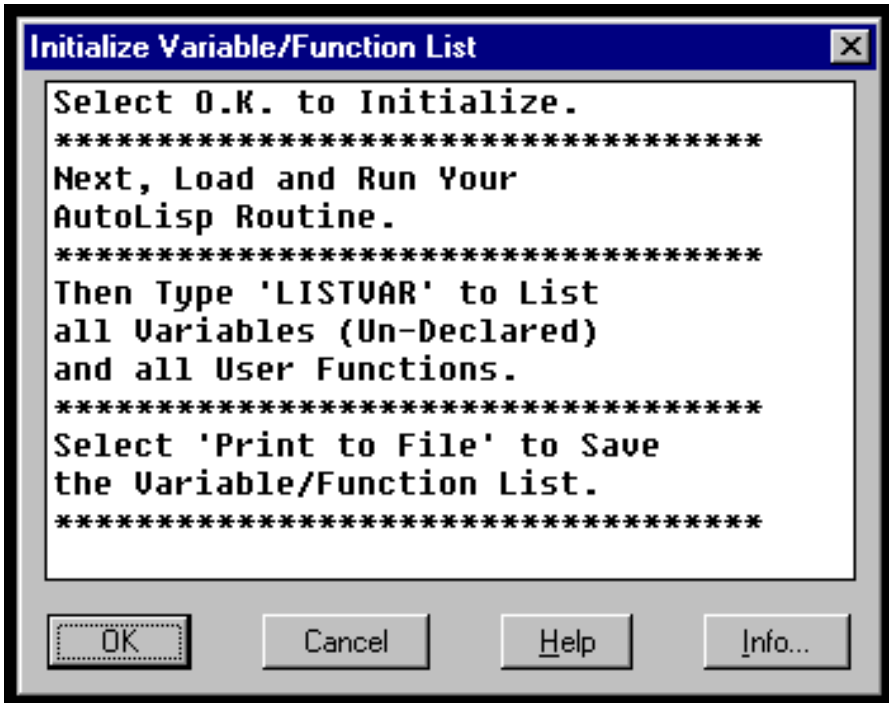
It is very important when writing AutoLISP routines to declare all your Local Variables. This prevents your variables stepping on other variables of the same name defined from other routines.
It often happens though, that on completion of your application, it can be quite difficult to remember and locate all the variables that you have used. This application will list all the un-declared variables and all functions used within your routine.

If you need help with Functions and Local/Global Variables, then refer to the DEFUN Function Tutorial.

Usage

I have included two sample AutoLISP files with this application, to demonstrate how to run and use 'ListVar.Lsp.

First of all, ensure that ListVar.Lsp and ListVar.Dcl are in your AutoCAD support path.
Now type (load "ListVar") at the command prompt.
A message will appear prompting you to type INIT to run the application.
Do just that. A dialogue box will appear that looks like this :



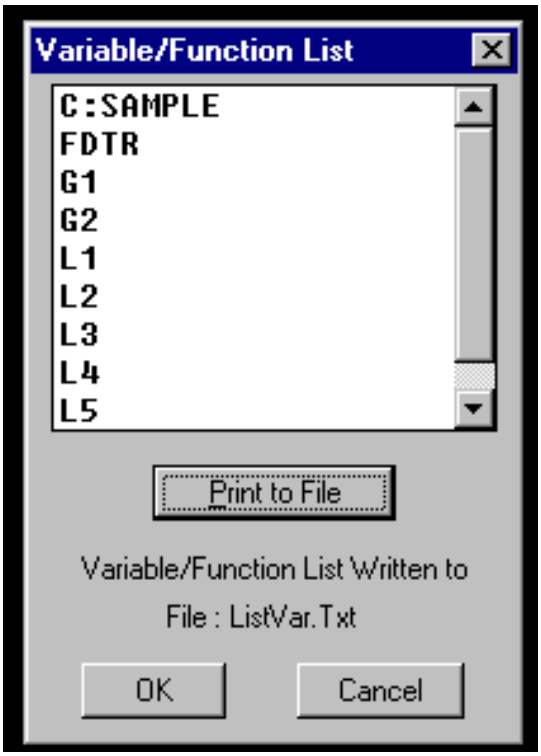
Read the instructions carefully.
Now load and run Sample.Lsp. The AutoLisp coding is as follows :

```
(defun c:sample ()

        (setq l1 10.0
              l2 20.0
              l3 30.0
              l4 40.0
              l5 50.0
              l6 (fDTR 45)
              g1 "Global1"
              g2 "Global2"
        );setq

    (princ)

);defun

(defun fDTR (x)

        (* PI (/ x 180.0))
);defun

(princ)
```

This is a very simple routine that basically defines 8 variables, 6 of which we want to declare Local and 2 of which we would like to declare as Global. We have also defined a sub-function that converts degrees to radians.

After running this routine type 'ListVar' at the command line.

Another dialogue will appear that looks like this :



As you can see, all the functions and variables used within the routine are listed in the list box.
To keep a permanent record of the list, select the 'Print to File' button.
This will print the list to the file 'ListVar.Txt :

```
C:SAMPLE
FDTR
G1
G2
L1
L2
L3
L4
L5
L6
```

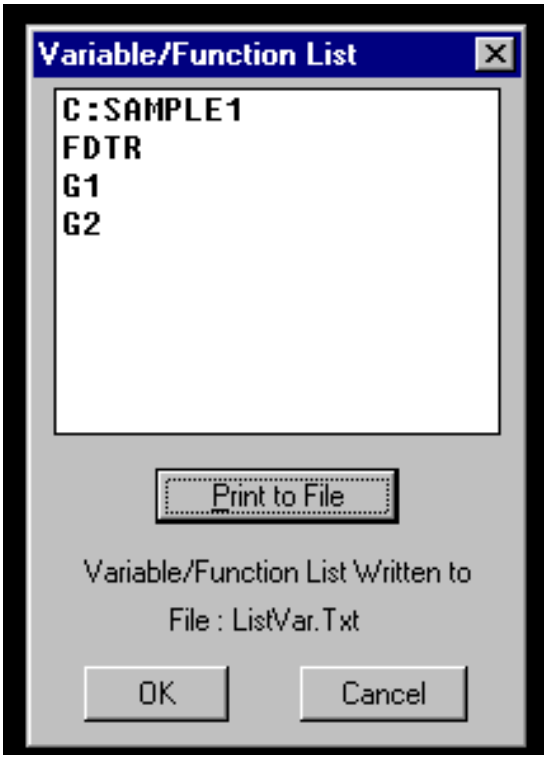
You can use this file to cut and paste the variables that you want to declare as Local into your AutoLisp file.

Now, re-run 'INIT' then load and run Sample1.Lsp.
Here is the coding for Sample1.Lsp.

```
(defun c:sample1 ( / L1 L2 L3 L4 L5 L6)
```

```
        (setq l1 10.0
              l2 20.0
              l3 30.0
              l4 40.0
              l5 50.0
              l6 (fDTR 45)
              g1 "Global1"
              g2 "Global2"
        );setq

    (princ)

);defun

(defun fDTR (x)

        (* PI (/ x 180.0))
);defun

(princ)
```

As you can see, we have declared all the Local Variables.
Now run 'ListVar' again. The dialogue should now look like this.



Do you see that the Local Variables that we declared are now not listed.
Using this application you can produce a list of all local variables that need to be declared

when you have written a new AutoLisp routine or, you can check your existing routines for variables that you may have missed.

Lisp Help.

When writing any sort of AutoLisp routine, one should supply some kind of Help. An Information box is also a handy thing to include as it allows you to inform the user of what version of the routine he is using, as well as giving you a chance to advertise yourself and your applications.
You can access and integrate custom Help into the Windows Help Files, but this is quite an involved process and you need the brains of a rocket scientist to accomplish this.

A much simpler way is to call an external Help or Info file from your AutoLisp routine. Using the following routine, if you click on the Help button, your default browser will open, displaying an HTML Help file.
Clicking on the Info button, will open Notepad and display an Info Text file.

Note: Please ensure that all files are in your AutoCad search path.

Here's the DCL Coding :

```
lisphelp : dialog {
           label = "Lisp Help Tutorial" ;

           ok_cancel_help_info ;

        }
```

And now the AutoLisp Coding :

```
;;;  ------------------------------------------------------------

(prompt "\n Type LISPHELP to run....")

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

(defun C:LISPHELP (/ USERFLAG1 USERFLAG2 USERFLAG3 HE INF DCL_ID)

  (setq DCL_ID (load_dialog "lisphelp.dcl")

  ) ;_ end of setq

  (if (not (new_dialog "lisphelp" DCL_ID)
```

```
    ) ;_ end of not
  (exit)
) ;_ end of if
(action_tile
  "accept"
  "(done_dialog) (setq userflag T)"
) ;_ end of action_tile
(action_tile
  "cancel"
  "(done_dialog) (setq userflag nil)"
) ;_ end of action_tile
(action_tile
  "help"
  "(done_dialog) (setq userflag1 T)"
) ;_ end of action_tile
(action_tile
  "info"
  "(done_dialog) (setq userflag2 T)"
) ;_ end of action_tile
(start_dialog)
(unload_dialog DCL_ID)
(if USERFLAG
  (alert "You selected O.K.")
) ;_ end of if
(if USERFLAG1
  (HEL)
) ;_ end of if
```

```
  (IF USERFLAG2
    (IN)
  ) ;_ end of IF
  (princ)
) ;_ end of defun
;;;  ------------------------------------------------------------------
(defun hel ()
  (if (not
        (setq HE (findfile "Help.htm")
        ) ;_ end of setq
      ) ;_ end of not
    (alert "Sorry - Help File Missing")
    (command "Browser" HE)
  ) ;_ end of if
  (princ)
) ;_ end of defun
(defun in ()
  (if (not
        (setq INF (findfile "Info.txt")
        ) ;_ end of setq
      ) ;_ end of not
    (alert "Sorry - Info File Missing")
    (startapp "notepad.exe" INF)
  ) ;_ end of if
  (princ)
) ;_ end of defun
(princ)
```

```
;;;EOF
;;;  ----------------------------------------------------------------
```

DOSLib - DOS Library Programmers Reference

I don't know if you are aware, but there is an application included with AutoCAD that makes dealing with the operating system a breeze when working with AutoLisp. This is DOSLib.Arx, designed and distributed by Robert Mcneel and Associates.

DOSLib, or DOS Library, is a library of LISP-callable functions that provide Windows operating system and DOS command-line functionality to various CAD applications, including AutoCAD and IntelliCAD.
DOSLib extends their LISP programming languages by providing the following functionality:

- Drive handling functions to change between drives and check disk space.
- Path handling functions to manipulate path specifications.
- Directory handling functions to create, rename, remove, select and change directories.
- File handling functions to copy, delete, move, rename, and select files. Functions for getting directory listings, searching and finding multiple instances of files, and changing attributes are provided.
- Print handling function to get and set default printers, and spool files.
- Initialization file handling functions to manipulate Windows-style initialization (INI) files, and Windows Registry access functions.
- Process handling functions to run internal DOS commands or other programs.
- Miscellaneous functions, like changing the system date and time, and displaying Windows message boxes.

In this Tutorial we will have a quick look at how to use some of the various functions available within DOSLib.

Creating a "Splash Screen"

This is a great way to promote yourself or your company.

DOSLib Function : dos_splash

Syntax : (dos_splash *filename duration*)

Arguments :

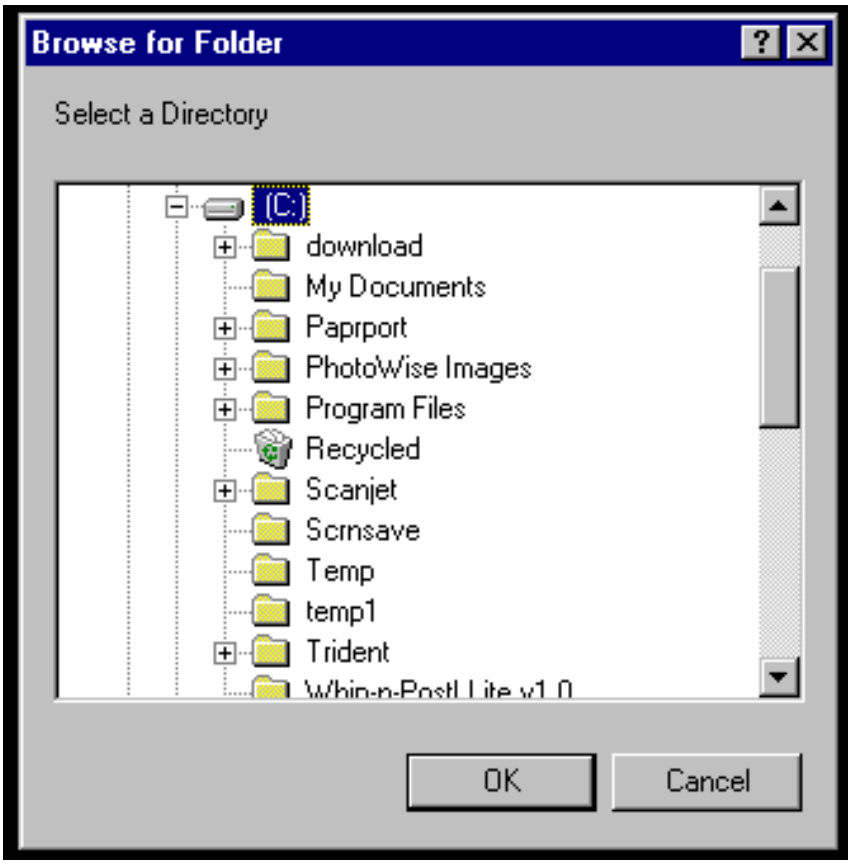
- *filename* - A 256-colour Windows BMP File.
- *duration* - Duration Time in Seconds.

Returns : nil if successful or on error.

Example :

```
(dos_splash "afralisp.bmp" 5)
```

Getting a Directory Path



This will display a Windows Browse for Folder Dialogue box and will return a fully qualified path to the selected directory.

DOSLib Function : dos_getdir

Syntax : (dos_getdir *title [path]*)

Arguments :

- *title* - A dialogue box title.

Options :

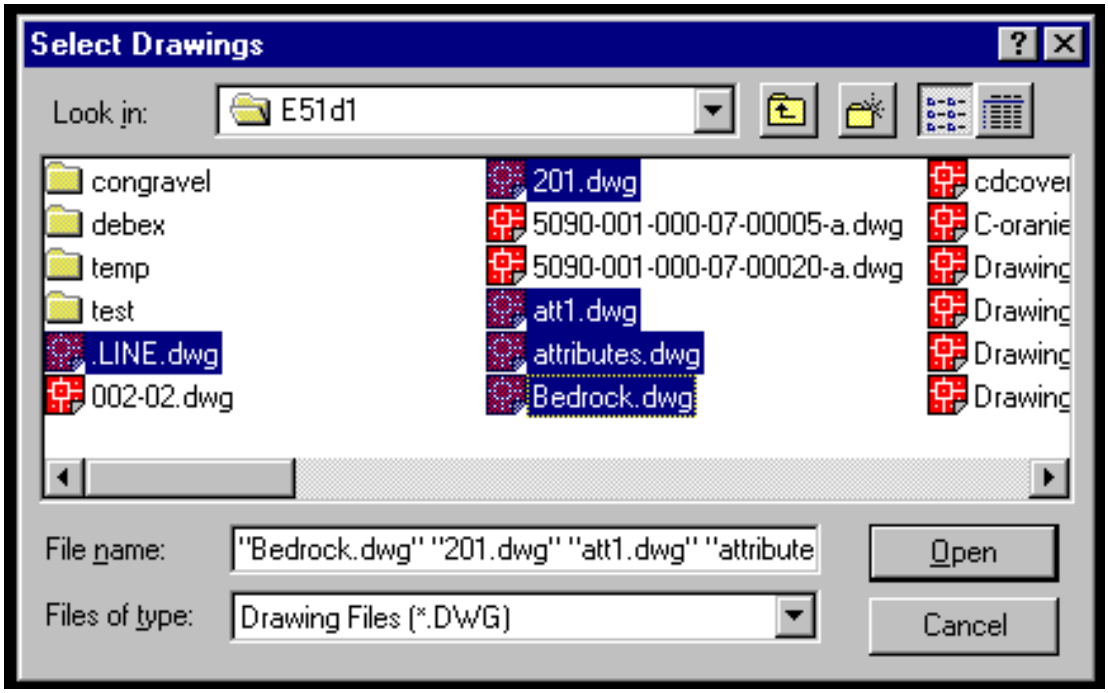
- *path* - An existing directory.

Returns : A qualified path to the current directory selected by the user. nil on cancel or error.

Example :

```
(setq udir (dos_getdir "Select a Directory" "c:\\"))
```

Multiple File Selection



Displays a Windows common file open dialogue box that will allow you to select multiple files.

DOSLib Function : dos_getfilem

Syntax : (dos_getfilem *title path filter*)

Arguments :

- *title* - A dialogue box title.
- *path* - An existing directory.
- *filter* - A filename filter string. The filter string consists of two components: a description (for example, "Text File"), and a filter pattern (for example, "*.txt"). Multiple filter patterns can be specified for a single item by separating the filter-pattern strings with a semi-colon (for example, "*.TXT;*.DOC;*.BAK"). The components must be separated by a pipe character ("|"). The filename filter string can consist of one or more filter strings, each separated by a pipe character.

Returns : A list of filenames. The first element of the list is a qualified path to

the selected directory. nil on cancel or error

Example :

```
(setq ufiles (dos_getfilem "Select Drawings" "c:\\drawings\\"
"Drawing Files (*.DWG)|*.DWG"))
```

AutoCAD Message Box

This displays a Windows Message Box. Much more flexible than the standard AutLisp "Alert" function. Very similar to the VBA Message Box function.

DOSLib Function : dos_msgbox

Syntax : (dos_getfilem *text title button icon*)

Arguments :

- *text* - The message to be displayed.
- *title* - The message box title.
- *button* - The push button format.
  The allowable values are :
  - 0 - Abort, Retry and Ignore.
  - 1 - OK.
  - 2 - OK and Cancel.
  - 3 - Retry and Cancel.
  - 4 - Yes and No.
  - 5 - Yes, No and Cancel.
- *icon* - The icon.
  The allowable values are :
  - 0 - Asterisk.
  - 1 - Exclamation.
  - 2 - Hand.
  - 3 - Information.
  - 4 - Question.
  - 5 - Stop.

Returns : The return values of dos_msgbox are as follows :

- 0 - Abort
- 1 - Cancel
- 2 - Ignore
- 3 - No
- 4 - OK
- 5 - Retry
- 6 - Yes

nil on error.

Example

```
(dos_msgbox "Continue Batch Process?" "Batch Purge" 5 4)
```

These are just a couple of examples of some of the functions available within DOSLib. There are 72 callable functions covering all areas of DOS usage. On the next page we'll have a look at an AutoLisp application designed using the DOSLib. See you there.....

DOSLib - Batch Slide Application

In this application we are going to use some of the DOSLib functions to Convert a selection of drawings to slides. I've explained the program within the coding by including comments for each line. Good luck and here we go...

```
(defun c:batchslide ( / CNTR DPATH DWGPATH FILE1 FLAG FLEN FNAME
                        FNAME1 LEN1 MESSAGE NFILES NFILES1 SCRFILE
                        UFILES)

;;;Firstly we need to check if DOSLib is loaded.
        (cond
          ((= (atoi (substr (getvar "acadver") 1 2)) 13)
            (if (not (member "doslib13.arx" (arx)))
              (arxload (findfile "doslib13.arx"))))
          ((= (atoi (substr (getvar "acadver") 1 2)) 14)
            (if (not (member "doslib14.arx" (arx)))
              (arxload (findfile "doslib14.arx"))))
          ((= (atoi (substr (getvar "acadver") 1 2)) 15)
            (if (not (member "doslib2k.arx" (arx)))
              (arxload (findfile "doslib2k.arx"))))
        );cond
;;;This, first checks the version of AutoCAD in use.
;;;It then checks to see if that version of Doslib is loaded.
;;;If it is not, it searches the AutoCAD search path for the
;;;relevant version, and then loads it.

;;;We must now check that, if we are using A2K, Single Document
;;;Mode is switched OFF.

        (if (= (atoi (substr (getvar "acadver") 1 2)) 15)
        ;if A2K
                (setvar "sdi" 0)
                ;switch off single document interface

        );if

;;;Now we are going to do a bit of advertising and display a
;;; Splash Screen for about 5 seconds.

        (dos_splash "afralisp.bmp" 5)
        ;display the splash screen

;;;Now, we need to retrieve the names of the files that we would
;;;like to make slides from.

        (setq dwgpath (getvar "dwgprefix"))
        ;get the path to the current directory
```

```
        (setq ufiles (dos_getfilem "Select Drawings"
        dwgpath "Drawing Files (*.DWG)|*.DWG"))
        ;display the file dialogue box

        (if (not ufiles)
        ;if no files selected or Cancel

                (exit)
                ;exit the application

        );

;;;The first item in the list "ufiles" is the directory path
;;;The remaining items are the file names.
;;;First, let's retrieve the directory path.

        (setq dpath (nth 0 ufiles))
        ;retrieve the directory path - the first item.

;;;Next, let's get the number of items in the list,
;;;and make sure that the user wants to continue.

        (setq nfiles (length ufiles))
        ;get the length of the list

        (setq nfiles1 (itoa (- nfiles 1)))
        ;count only the file names and convert to a string

        (setq message (strcat "You have choosen " nfiles1 " files.
                              \nDo you want to continue?"))
        ;set up the message

        (setq flag (dos_msgbox message "AfraLisp Batch Slides" 4 4))
        ;display the message box

        (if (= flag 6)
        ;if Yes selected

        (progn
        ;do the following

;;;We'll now open an external file to write our script to.

        (setq scrfile (strcat dpath "batdir.scr"))
        ;add the path to the script file name

        (setq file1 (open scrfile "w"))
        ;open the file to write

        (setq cntr 1)
        ;set the counter
```

```
;;;We'll now start the loop, format the file names, and add
;;;the commands to the script file.

        (repeat (- nfiles 1)
        ;start the loop

                (setq fname (nth cntr ufiles))
                ;get the file name

                (setq fname (strcat dpath fname))
                ;add the path to the file name.
                ;"fname" will be used to open each
                ;drawing.

                (setq flen (strlen fname))
                ;get the length of file name

                (setq len1 (- flen 4))
                ;take away the last 4 characters (.DWG)

                (setq fname1 (substr fname 1 len1))
                ;get just the filename without the extension.
                ;"fname1" will be used as our slide file name.

;;;write the commands to the script file

                (write-line (strcat "open " fname) file1)
                ;open the drawing
                ;write it to the script file

                (write-line (strcat ".zoom" " e") file1)
                ;zoom to extents
                ;write it to the script file

                (write-line (strcat "filedia" " 0") file1)
                ;switch off dialogues
                ;write it to the script file

                (write-line (strcat "mslide " fname1) file1)
                ;make the slide
                ;write it to the script file

                (write-line (strcat "filedia" " 1") file1)
                ;switch on dialogues
                ;write it to the script file

                (write-line "qsave" file1)
                ;save the drawing
                ;write it to the script file
```

```
                (if (= (atoi (substr (getvar "acadver") 1 2)) 15)
                ;if A2K

                (write-line "close" file1)
                ;close the drawing
                ;write it to the script file

                );if

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

        );repeat

;;;now that we've finished writing the commands to the script
;;;file, we must remember to close it.

        (close file1)
        ;close the script file

        (command "script" scrfile)
        ;run the script file and sit back and have a beer.

        );progn

        );if

 (princ)

);defun

(princ)
```

The script file, *batdir.scr* should look something like this :

```
open O:\E51D\E51D1\temp\L102.dwg
.zoom e
filedia 0
mslide O:\E51D\E51D1\temp\L102
filedia 1
qsave
close  (A2K Only)
open O:\E51D\E51D1\temp\L101.dwg
.zoom e
filedia 0
mslide O:\E51D\E51D1\temp\L101
filedia 1
qsave
close  (A2K Only)
```

AfraLisp Slide Manager

Creating Icon menus's in AutoCAD can be quite tedious. First you need to create slides of all your drawings that you want to include in your menu. Secondly, you need to create a text file with the names of all the slides. Thirdly you need to run "Slidelib.exe" to create your slide menu. And last but not least, you need to add all the menu items and macros to your menu. A lot of work hey! Then, 2 months later you want to add something to the menu and you have to go through the whole process again.

This is where the "AfraLisp Slide Manager" comes in. This application will allow you to create Slide Libraries without the overhead of having to create slide libraries or modify your menu files.

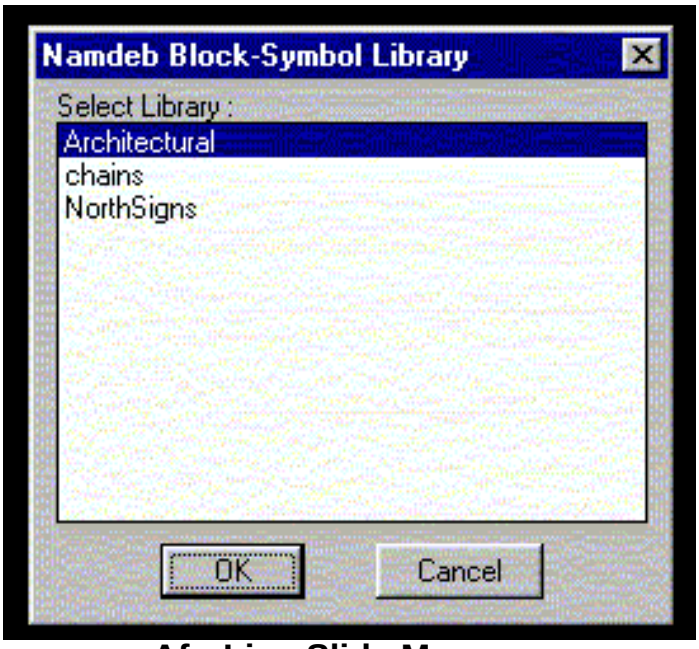
Usage :

Create a root directory (eg. C:/Slidemanager) ensuring that Slideman.Lsp, Slideman.Dcl and Filepath.Dat are within this directory.
Important : This directory MUST be added to your AutoCAD Support Path.

Next, create as many sub-directories as you wish, placing your drawing files and slide files within these directories. (eg. C:/Slidemanager/Architecural; C:/Slidemanager/Chains; C:/Slidemanager/NorthSigns; etc.)

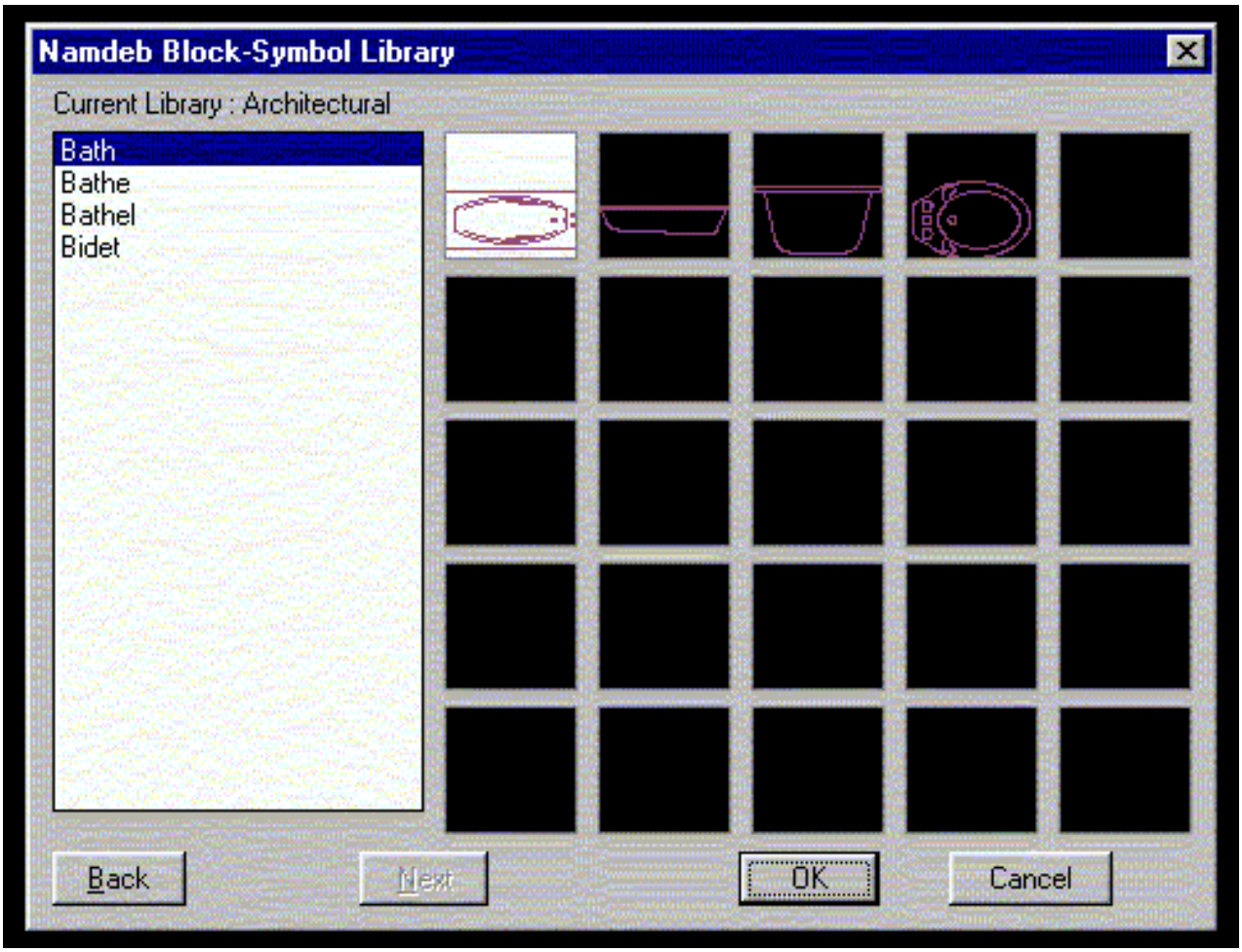
Now, load and run "Slideman.Lsp"

The first dialogue is where you choose which library you wish to open.



AfraLisp Slide Manager

The next dialogue is where you choose which block to insert.



Anytime you wish to add new drawings to the slide library, just add them to an existing library, or create a new one.

The only restriction is that only 75 drawings are allowed in any one library. (This excludes the slides.)

Working With Areas

The easiest way of calculating areas in AutoCAD is to simply use the "Area" command. This command though, leaves a lot to be desired when in comes to usage. And the resulting format? Well I'm not going to even mention that. (Hang on, you've got to. That's what this tutorial is all about.) Oop's, silly me.
As you have probably noticed, when you use the "Area" command, it returns the result in the square units of whatever units you are drawing in. eg. If you are drawing in millimeters, it returns the result in square millimeters. No more, and no less.
Now this may suit some folk, but me, no way. Even though I work in millimeters, I want the result returned in square meters AND I want to be able to place the area as text, anywhere in the drawing that suits me AND nicely formatted with a little square meter thingymebob at the end .
So, what do we do? Easy, we open notepad and start banging away on the keyboard.

```
;;;M2 Lisp - Version 1.0 - 2nd August 2001
;;;============================================================
;;;This function will calculate the area (m2) from
;;;points input by the user. The user then has the
;;;option of placing a text label into the drawing using
;;;the current text style/height at a user defined
;;;insertion point.
;;;============================================================
;;;Written by Kenny Ramage August 2001
;;;============================================================
;;;============================================================
;;;Define Main Function
;;;============================================================

(defun C:M2 ( / os oom laag oec oudosmode p q opp oppm oppma oppmat
                tekst pos pos2 flag1 antw lw)

   (setvar "cmdecho" 0)

   (command "undo" "m")

   (setq oom (getvar "orthomode")
         laag (getvar "clayer")
         oudosmode (getvar "osmode")
         olderr *error*
         *error* opperr
   );setq

   (setvar "orthomode" 0)
   (print)
   (prompt "\nArea Lisp V-1.0 Written by Kenny Ramage
```

- kramage@mweb.com.na") (prompt "\nPick the corners of the area you want to calculate : ")

(initget 1) (setq p (getpoint "\nFrom point : ") q (list (car p)(cadr p)));setq (command "layer" "m" "5" "" "pline" q "w" "0" "");command (while (= (getvar "cmdnames") "PLINE") (redraw (entlast) 3) (prompt "\nTo point [Close/Undo] : ") (command pause));while (command "area" "e" "l") (setq opp (getvar "area") oppm (/ opp 1000000.0) oppma (rtos oppm 2 3) oppmat (strcat oppma "m") tekst (strcat "\nArea = " oppmat "2"));setq (setq lw (entlast)) (command "layer" "m" laag "") (prompt tekst) (setq flag1 T) (while flag1 (setq antw (getstring "\nInsert Area Label? [y/n] : ")) (setq antw (strcase antw)) (if (or (= antw "Y")(= antw "")) (progn (if (not (tblsearch "layer" "4")) (command "layer" "m" "4" "c" "4" "4" "") (command "layer" "t" "4" "on" "4" "u" "4" "s" "4" ""));if (setvar "osmode" 0) (setq pos (getpoint "\nInsertion Point : ")) (if (= (cdr (assoc 40 (tblsearch "style" (getvar "textstyle")))) 0) (command "text" "j" "c" pos "" "0" oppmat) (command "text" "j" "c" pos "0" oppmat));if (setq pos2 (cadr (textbox (entget (entlast)))) pos2 (list (+ (car pos)(/ (car pos2) 2.0))

```
                        (+ (cadr pos)(cadr pos2)))
            );setq

            (if (= (cdr (assoc 40 (tblsearch "style"
                (getvar "textstyle")))) 0)
                (command "text" "j" "tl" pos2 "" "0" "2")
                (command "text" "j" "tl" pos2 "0" "2")
            );if

            (command "scale" "l" "" pos2 ".5")

        );progn

    );if

         (if (or (or (= antw "Y")(= antw "N")(= antw "")))

                 (setq flag1 nil)
         );if

    );while

    (command "erase" lw ""
             "redrawall"
    );command

;;;================================================
;;;Reset System Variables and Restore Error Handler
;;;================================================

    (setq *error* olderror)
    (command "layer" "m" laag "")
    (setvar "osmode" oudosmode)
    (setvar "orthomode" oom)

    (princ)
);defun C:M2
```

```
;;;=================================================
;;;Define Error Trap
;;;=================================================

(defun opperr (s)

      (if (/= s "Function cancelled")
          (princ (strcat "\nError: " s))
      );if

      (setq *error* olderr)
      (setvar "osmode" oudosmode)
      (setvar "orthomode" oom)
      (command "undo" "b"
               "layer" "m" laag ""
               "redrawall"
      );command

);defun opperr
;;;=================================================
(princ)
;;;End Area Lisp
;;;=================================================
;;;=================================================
```

Load the application and then type "M2" to run. Select the area that you would like to calculate by picking selected points. A polyline will be traced from each point that you have selected. When you have finished, select "C." (Close.)
The area will now be displayed at the command line. You will now be given the option to add a label to your drawing if you so wish.

But now we'll get really clever and try and fool AutoCAD into working out the area that we want automatically. Interested? Read on.....

Nice to see you back and glad that you could make it.
Where were we? Oh yes. We were going to try and make AutoCAD determine the area to be calculated. Let's see! We could try bribes! We could threaten!
Naaw, back to the keyboard.....

```
;;;M2 Lisp - Version 1.0 - 17th October 2001
;;;============================================================
;;;This function will calculate an irregular area (m2)
;;;using boundary detection. The user then has the
;;;option of placing a text label into the drawing using
;;;the current text style/height at a user defined
;;;insertion point.
;;;============================================================
;;;Written by Kenny Ramage October 2001
;;;============================================================
;;;============================================================
;;;Define Main Function
;;;============================================================

(defun C:M2A ( / os oom laag oec oudosmode p q opp opp1 oppm

oppma oppmat tekst pos pos2 flag1 antw lw a b) (setvar "cmdecho" 0) (command "undo" "m") (setq oom (getvar "orthomode") laag (getvar "clayer") oudosmode (getvar "osmode") olderr *error* *error* opperr );setq (setvar "orthomode" 0) (print) (prompt "\nIrregular Area Lisp V-1.0 Written by Kenny Ramage - kramage@mweb.com.na") (setq opp 0.0) (command "Layer" "m" "2" "") (while (setq a (getpoint "\nSelect Internal Point: ")) (command "-Boundary" a "") (setq b (entlast)) (redraw b 1) (command "area" "O" "L") (setq opp1 (getvar "area")) (setq opp (+ opp opp1)) ;(redraw b 4) );while (setq oppm (/ opp 1000000.0) oppma (rtos oppm 2 3) oppmat (strcat oppma "m") tekst (strcat "\nArea = " oppmat "2") );setq (command "layer" "m" laag "") (prompt tekst) (setq flag1 T) (while flag1 (setq antw (getstring "\nInsert Area Label? [y/n] : ")) (setq antw (strcase antw)) (if (or (= antw "Y")(= antw "")) (progn (if (not (tblsearch "layer" "4")) (command "layer" "m" "4" "c" "4" "4" "") (command "layer" "t" "4" "on" "4" "u" "4" "s" "4" "") );if (setvar "osmode" 0) (setq pos (getpoint "\nInsertion Point : ")) (if (= (cdr (assoc 40 (tblsearch "style" (getvar "textstyle")))) 0) (command "text" "j" "c" pos "" "0" oppmat) (command "text" "j" "c" pos "0" oppmat) );if (setq pos2 (cadr (textbox (entget (entlast)))) pos2 (list (+ (car pos)(/ (car pos2) 2.0)) (+ (cadr pos)(cadr pos2))) );setq (if (= (cdr (assoc 40 (tblsearch "style" (getvar "textstyle")))) 0) (command "text" "j" "tl" pos2 "" "0" "2") (command "text" "j" "tl" pos2 "0" "2") );if (command "scale" "l" "" pos2 ".5") );progn );if (if (or (or (= antw "Y")(= antw "N")(= antw ""))) (setq flag1 nil) );if );while
;;;============================================================ ;;;Reset System Variables and Restore Error Handler
;;;============================================================ (setq *error* olderror) (command "layer" "m" laag "") (setvar "osmode" oudosmode) (setvar "orthomode" oom) (princ) );defun C:M2A ;;;============================================================
;;;Define Error Trap ;;;============================================================
(defun opperr (s) (if (/= s "Function cancelled") (princ (strcat "\nError: " s)) );if (setq *error* olderr) (setvar "osmode" oudosmode) (setvar "orthomode" oom) (command "undo" "b" "layer" "m" laag "" "redrawall" );command );defun opperr
```

**;;;========== (princ) ;;;End Area Lisp ;;;==========
;;;========== This time, select a point within the area that you would like to calculate. A boundary box will be drawn around the area you have chosen. You can add to the area if you wish by selecting more points resulting in an accumulation of areas. Again, you have the option of a label being added.**

AutoCAD and HTML

This time we're going to have a look at creating an HTML file that lists a directory of DWF drawings using AutoLisp. To select the DWF files and the relevant directory, we are going to make use of a very powerful DosLib function "*dos_filem.*" Oh, before I forget, you will need to have the DosLib library installed on your workstation. If you don't have DosLib, you can download the latest version from Robert McNeel & Associates.

I'd like to thank Andy Canfield(Cornbread) for his ideas and input into this routine. Andy wrote about 90% of this program and in fact, gave me the initial idea for this article. I've changed a couple of things and tidied it up for publishing, but Andy did the major work. Any errors or typos are from my desk. Right, enough waffle, lets get on with it. Copy and paste this into Notepad and then save it as "Dwf-Html.lsp."

```
(defun C:Dwf-Html ( / flag thelist thedir nfiles thefile fn ctr dname)

;set the flag
(setq flag T)

;check Doslib is loaded
(if (not (member "doslib2k.arx" (arx)))

  (progn

    (if (findfile "doslib2k.arx")

      (arxload "doslib2k")

    (progn

      (alert "DosLib not installed")

      (setq flag nil)

    );progn

   );if

  );progn

);if

;if DosLib is installed and loaded
(if flag

;do the following
(progn

  ;select the DWF files
  (setq thelist (dos_getfilem "Select Drawings"
                        "C:\\" "DWF Files (*.dwf)|*.dwf"))

  ;retrieve the directory
  (setq thedir (car thelist))

  ;retrieve the file names
  (setq thelist (cdr thelist))
```

```
  ;get the number of files
  (setq nfiles (length thelist))

  ;create the HTML file
  (setq thefile (strcat thedir "Dwf-Html.html"))

  ;open the HTML file
  (setq fn (open thefile "w"))

  ;write the header information
  (write-line "<html><head>
               <title>AutoCAD and HTML</title>
               </head><body>" fn)

  ;give it a title
  (write-line "<h1>AutoCAD and HTML</h1><hr>" fn)

  ;set the counter
  (setq ctr 0)

  ;start the loop
  (repeat nfiles

     ;get the drawing name
     (setq dname (nth ctr thelist))

     ;create the HTML link
     (write-line (strcat "<a href =" "\""
                         thedir dname "\"" ">" thedir dname "
                         </a><br>") fn)

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

  );end repeat

  ;write the HTML footer
  (write-line "<hr><h3>Brought to you by CAD Encoding</h3>
               </body></html>" fn)

  ;close the file
  (close fn)

  ;inform the user
  (alert (strcat "File saved as" "\n" thedir "Dwf-Html.html"))

  );progn

 );if flag

  ;finish clean
  (princ)

);defun

;load clean
(princ)
```

Now, load and run this routine.
A file dialog will appear. Choose the directory your DWf files are located in, and then select the DWF files you would like listed in the HTML report.
The HTML file will be created and stored in the same directory as your DWF files and will be named "Dwf-Html.html." Open this file in your browser. You should have a list of links to all the DWF files you selected.

As I mentioned earlier, we make use of the DosLib function "*dos_filem*" to allow the user to select the DWF files that he/she/it wants to process.
The "*dos_filem*" function displays a file open dialog box that allows for multiple file selection. The function returns a list of filenames if successful. The first element of the list is a qualified path to the selected directory. eg.

(setq thedir (car thelist))

returns the file directory and

(setq thelist (cdr thelist))

returns a list of file names. Do yourself a favour and read the help file that comes with DosLib as you'll find a lot of useful functions that will make your AutoLisp coding a lot easier.

AutoCAD and HTML - Revisited

I received this from Andy Canfield :

Kenny,
Since you are so nice I am attaching my super cool version of your function which allows an onclick event. Switches the direction of the slashes and produces a formatted link. Very cool.
Andy.

```
(defun C:Dwf-Html ( / flag thelist thedir nfiles
                      thefile fn ctr dname flen
                      len1 shortdname revSlash
                      f_oldstr f_newstr f_string)

   ;set the flag
   (setq flag T)

   ;check Doslib is loaded
   (if (not (member "doslib2k.arx" (arx)))

      (progn

         (if (findfile "doslib2k.arx")

            (arxload "doslib2k")

            (progn

               (alert "DosLib not installed")

               (setq flag nil)

            );progn

         );if

      );progn

   );if

   ;if DosLib is installed and loaded
   (if flag

      ;do the following
      (progn

         ;select the DWF files
         (setq thelist (dos_getfilem "Select Drawings"
```

```
        "C:\\" "DWF Files (*.dwf)|*.dwf"))

        ;retrieve the directory
        (setq thedir (car thelist))

        ;need to reverse the slashes
        (setq f_oldstr "\\")
        (setq f_newstr "\/")
        (setq f_string thedir)
        (setq revSlash (replace f_oldstr
                       f_newstr f_string))

        ;retrieve the file names
        (setq thelist (cdr thelist))

        ;get the number of files
        (setq nfiles (length thelist))

        ;create the HTML file
        (setq thefile (strcat thedir
                       "Dwf-Html.html"))

        ;open the HTML file
        (setq fn (open thefile "w"))

        ;set the counter
        (setq ctr 0)

        ;start the loop
        (repeat nfiles

           ;get the drawing name
           (setq dname (nth ctr thelist))

           ;get the length of the drawing name
           (setq flen (strlen dname))

           ;length of drawing name minus .dwf
                       (setq len1 (- flen 4))

                       ;the drawing name minus .dwf
                       (setq shortdname (substr dname 1 len1))

           ;create the HTML link
           (write-line (strcat "<a href=# onClick=" "\""
               "window.open(" "'" "file:" revSlash dname "'" ",
               'modifications','toolbar=no,menubar=no,
               status=yes,resizable=yes,scrollbars=yes,
               width=800,height=600'); return false;
               " "\"" ">" shortdname "</a><br>") fn)
```

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

          );end repeat

          ;close the file
          (close fn)

          ;inform the user
          (alert (strcat "File saved as" "\n"
                 thedir "Dwf-Html.html"))

       );progn

    );if flag

    ;finish clean
    (princ)

);defun

;load clean
(princ)

;the following function was Written By Michel Loftus
(defun replace (f_oldstr f_newstr
                f_string / f_cur f_loop)
;like subst but with strings not lists
  (if (null (and (stringp f_oldstr)
                 (stringp f_newstr)
                 (stringp f_string)
            )
      )
    (exit)
  )
  (if (> (strlen f_oldstr) (strlen f_string))
    (exit)
  )
  (setq f_loop 1)
  (repeat (strlen f_string)
    (progn
      (setq f_cur (substr f_string f_loop
        (strlen f_oldstr)))
      (if (= f_cur f_oldstr)
        (setq f_string (strcat (substr f_string 1 (- f_loop 1))
                               f_newstr
                               (substr f_string
                                       (+ f_loop (strlen f_oldstr))
                                       (strlen f_string)
```

```
                                    )
                            )
                )
            )
            (setq f_loop (1+ f_loop))
        )
    )
    f_string
)
;eg (replace "\t" " " "Line one\tLine two") = "Line one Line two"

;the following function was Written By Michel Loftus
(defun stringp (f_val) ;test if a string
    (if (= (type f_val) 'STR)
        t
        nil
    )
)
;eg (stringp "Hello") = true
```

Environmental Issues

Setting up Support Paths within AutoCAD can be a real pain in the proverbial backside especially, if you've got a whole stack of workstations to deal with. It's even worse, if like my office, each user has different menu settings and support files.
The following coding shows you how you can automate this process by making use of the AutoCAD *(getenv)* and *(setenv)* functions.

Firstly we need to set up our Support Path directories. Mine looks like this :

C:/NDBE51D1
|
|
|---------- LISP
|---------- VBA
|---------- BLOCKS
|---------- DRAWINGS
|---------- MENU
|---------- TEMPLATES

NDBE51D1 is my login name. By incorporating the users login name into the support path, we know who we are dealing with and can set up our Support Paths to suit.

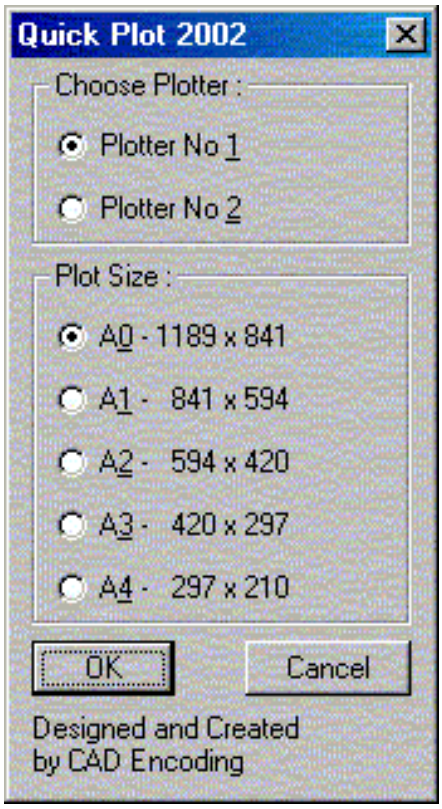
```
;CODING STARTS HERE

(defun C:SETNEWPATH ( / netpath defpath)

;define the network path - replace this with your
;network path
(setq netname "C:\\")

;define the user directory path
(setq netpath (strcat netname (getvar "LOGINNAME")))

;set to AutoCAD default
(setenv "ACAD" "")

;store the default paths
(setq defpath (getenv "ACAD"))

;set up the custom paths
(setenv "ACAD" (strcat
defpath ";"
netpath ";"
netpath "\\" "lisp;"
netpath "\\" "vba;"
netpath "\\" "blocks;"
netpath "\\" "drawings;"
netpath "\\" "menu;"
))
```

```
;set up the custom template path
(setenv "TemplatePath" (strcat netpath "\\" "Templates"))

;set up the custom temp directory path
(setenv "Temp" "C:\\temp")

;inform the user
(alert "Custom Support Paths Defined")

(princ)

);defun

(princ)

;CODING ENDS HERE
```

You could incorporate this into your AcadDoc.lsp file to ensure that all your Support Paths are set correctly. Irrespective of the user, the correct Support Path will be defined.

Quick Plot

Plotting in AutoCAD can be quite a time consuming and sometimes confusing exercise. In most drawing offices, there are normally not more than two plotters/printers, and if you look at things carefully, you will find that that the majority of the plots are produced using the same plot settings. As well, plotters only change at somewhat long intervals, therefore a wee bit of hard coding in regards to Plotter Configuration and Plot Styles is really neither here nor there.



That's where Quick Plot comes in. Quick Plot can be set up to use your more common plot configurations, plot styles and plot sheets, drastically cutting down on the time needed to plot.

Quick Plot is simple to use. Window the section of the drawing you want plotted - we use crop marks to ensure true scaling - select the plotter you want to plot too, select the size of sheet you would like to use, and away you go.

To customise this routine, all you have to do is create a Plot Configuration file (PC3) to suit your plotter, create a Plot Style file (CTB) and then replace the names of these files with the names of the Plotters and Plot Styles within the coding.

First the DCL coding. Save this as "Q-Plot.dcl."

```
qplot : dialog {
    label = "Quick Plot 2002";

     : boxed_radio_column {
      label = "Choose Plotter :";

        //change the plotter name to suit your own
        : radio_button {
          label = "Plotter No &1";
          key = "rb1";
        }
        //change the plotter name to suit your own
        : radio_button {
          label = "Plotter No &2";
          key = "rb2";
        }

     }


     : boxed_radio_column {
      label = "Plot Size :";

        : radio_button {
          label = "A&0 - 1189 x 841";
          key = "rb5";
        }
        : radio_button {
          label = "A&1 -   841 x 594";
          key = "rb6";
        }
        : radio_button {
          label = "A&2 -   594 x 420";
          key = "rb7";
        }
        : radio_button {
          label = "A&3 -   420 x 297";
          key = "rb8";
        }
        : radio_button {
          label = "A&4 -   297 x 210";
          key = "rb9";
        }
     }

      ok_cancel ;

                : paragraph {
                  : text_part {
                    label = "Designed and Created";
```

```
                }
                : text_part {
                  label = "by CAD Encoding";
                }
              }
}

choose : dialog {
    label = "Quick Plot 2002";

       : text {
       label = "Continue with Plot?";
       alignment = centered;
       }

       ok_cancel ;

}
```

And Now the AutoLisp coding. Save this as "Q-Plot.lsp."

```
;;;CODING STARTS HERE

;|To customise this to suit your plotters, you need to do 3 things :

Create a PC3 plot configuration file and replace the value of the
variable "pltyp" with the name of your PC3 file.
eg. Replace  "RW-470 PLOTBASE.pc3" with "YOURPLOTTER.pc3"

Create a CTB plot style and replace the value of the variable "plstyle"
with the name of your CTB file.
eg. Replace "acad.ctb" with "your.ctb"

Replace all the "plsize" variables with your sheet sizes.

================================================|;

(defun C:Q-PLOT ( / userclick userclick1 pltyp plstyle plsize
                                oldblip oldecho oldsnap dcl_id)

    ;preset the flags
    (setq userclick T)
    (setq userclick1 T)

    ;store system settings
    (setq oldblip (getvar "BLIPMODE")
            oldecho (getvar "CMDECHO")
            oldsnap (getvar "OSMODE"))
```

```
;set system variables
(setvar "BLIPMODE" 0)
(setvar "CMDECHO" 0)
(setvar "OSMODE" 32)

;store the default plotter, plot style and plot size

;replace this with your pc3 file
(setq pltyp "RW-470 PLOTBASE.pc3")
;replace this with your ctb file
(setq plstyle "acad.ctb")
;replace this with your default sheet size
(setq plsize "ISO A0 (1189.00 x 841.00 MM)")

;get the plotting area
(setq pt1 (getpoint "\nSelect Lower Left Hand Corner
            of Plotting Area: "))
(setq pt2 (getcorner pt1 "\nSelect Upper Right Hand
            Corner of Plotting Area: "))

;remember your manners
(prompt "\nThanks.......")

;load the dialog
(setq dcl_id (load_dialog "q-plot.dcl"))
(if (not (new_dialog "qplot" dcl_id) ) (exit))
(set_tile "rb1" "1")
(set_tile "rb5" "1")

 ; set the tiles

 ;replace this with your first pc3 file
 (action_tile "rb1"
    "(setq pltyp \"RW-470.pc3\")")

;replace this with your second pc3 file
(action_tile "rb2"
    "(setq pltyp \"DesignJet 700.pc3\")")

;replace all sheet sizes to match yours
(action_tile "rb5"
    "(setq plsize \"ISO A0 (1189.00 x 841.00 MM)\")")
(action_tile "rb6"
    "(setq plsize \"ISO A1 (841.00 x 594.00 MM)\")")
(action_tile "rb7"
    "(setq plsize \"ISO A2 (594.00 x 420.00 MM)\")")
(action_tile "rb8"
    "(setq plsize \"ISO A3 (420.00 x 297.00 MM)\")")
(action_tile "rb9"
    "(setq plsize \"ISO A4 (297.00 x 210.00 MM)\")")
```

```
    (action_tile "cancel"
       "(done_dialog)(setq userclick nil)")
   (action_tile "accept"
       "(done_dialog) (setq userclick T)")

   ;start the dialog
   (start_dialog)

   ;unload the dialog
   (unload_dialog dcl_id)

;if OK is selected
(if userclick

  ;do the following
  (progn

      ;preview the plot
      (command "Preview")

          ;load the "Continue" dialog
          (setq dcl_id (load_dialog "q-plot.dcl"))
          (if (not (new_dialog "choose" dcl_id) ) (exit))

          ;set up the tiles
          (action_tile "cancel"
          "(done_dialog)(setq userclick1 nil)")
          (action_tile "accept"
          "(done_dialog) (setq userclick1 T)")
          (start_dialog)
          (unload_dialog dcl_id)

     ;if it's OK to continue
     (if userclick1

      ;plot the drawing
      (command "-plot" "Yes" "Model" pltyp plsize "Millimeters"
               "Landscape" "No" "Window" pt1 pt2 "Fit" "Center"
               "Yes" plstyle "Yes" "No" "No" "Yes" "Yes")

      ;the plot was Cancelled
      ;return to the main dialog
      (c:autoplot)

     );if userclick1

     );progn

);if userclick

   ;reset system variables
   (setvar "BLIPMODE" oldblip)
```

```
    (setvar "CMDECHO" oldecho)
    (setvar "OSMODE" oldsnap)

 (princ)

);defun c:q-plot

;;;*============================================================
(princ)

;CODING ENDS HERE
```

Acknowledgments and Links

A big thanks to :

My wife Heather.
My Mum and Dad
***EVERYONE* at VBA Expresso.**
The lads from BedRock - Pete, Eddie and Mike.
Frank Zander for his generosity and support.

And a BIG thank you to Marie and Jessie Rath for at least trying to control that reprobate Randall. (Thanks for everything pal.)

If I've forgotten someone, hey life's tough.....

Some of the best Links on the Web :

AfraLisp - http://www.afralisp.com

VBA Expresso - http://www.vbdesign.net/cgi-bin/ikonboard.cgi

CAD Encoding - The Journal - http://www.cadencoding.com

VB Design - http://www.vbdesign.net

Contract CADD Group - http://www.contractcaddgroup.com

CopyPaste Code - http://www.copypastecode.com

DsxCad - http://www.dsxcad.com

QnA~Tnt - http://www.dczaland.com/appinatt

BedRock - http://www.bedrockband.com