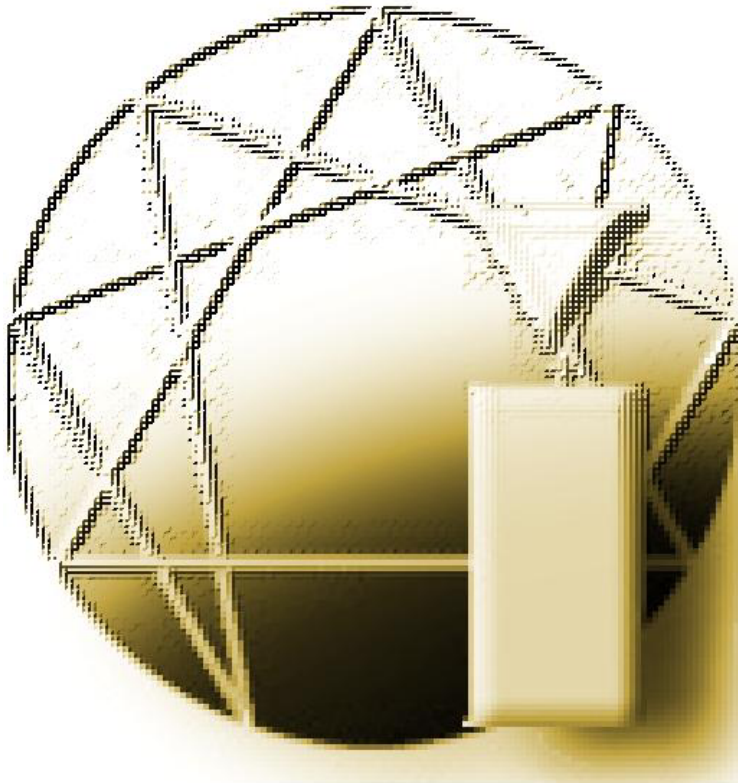


tailoring progeCAD



customizing and programming progeCAD
by R. H. Grabowski

upFront.eZine Publishing

Copyright Information

Copyright © 2009 by upFront.eZine Publishing, Ltd. All rights reserved worldwide. All Rights Reserved

All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

This book is sold as is, without warranty of any kind, either express or implied, respecting the contents of this book and any disks or programs that may accompany it, including but not limited to implied warranties for the book's quality, performance, merchantability, or fitness for any particular purpose. Neither the publisher, authors, staff, or distributors shall be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by this book.

Technical writer **Ralph Grabowski**

Technical editors **The progeCAD Team**

Copy editor **Herbert Grabowski**

Contents

1 • Tailoring the Environment of progeCAD 13

Starting progeCAD 2009	14
Changing the User Interface	14
Changes Through Windows	14
<i>Windows 2000 and XP</i>	<i>14</i>
<i>Windows Vista and 7</i>	<i>15</i>
Changes Through progeCAD	15
<i>Options: General</i>	<i>15</i>
Options: Paths	17
<i>Search Path Options</i>	<i>18</i>
<i>Default System File Names</i>	<i>20</i>
Options: Display	21
<i>Graphics Window</i>	<i>21</i>
<i>Background Colors</i>	<i>21</i>
<i>Menus</i>	<i>22</i>
Crosshair Colors and Size	23
<i>Axis Color</i>	<i>23</i>
Snap Cursor Colors and Markers	24
Other User Interface Elements	26
Command Bar	26
Status Bar	27
Drawing Tabs	27
Toolbars	28
Startup Options	28
Statup Switch	29
<i>/b Switch</i>	<i>29</i>

2 • Creating Keystroke Shortcuts & Aliases 30

Shortcut Keys	31
Defining Shortcut Keys	32
<i>Editing Keyboard Shortcuts</i>	<i>33</i>
<i>Deleting Keyboard Shortcuts</i>	<i>34</i>
Assigning Multiple Commands	34
Command Aliases	35
Creating New Aliases	36
<i>Editing Aliases</i>	<i>37</i>
<i>Deleting Aliases</i>	<i>37</i>
Rules for Writing Aliases	37
progeCAD Aliases: Sorted by Command Name	38

Sharing Shortcuts	40
Exporting Shortcuts & Aliases	40
Importing Shortcuts and Aliases	41
<i>Importing Through the Command Bar</i>	<i>41</i>
File Formats	42
Keystroke Shortcuts - .ick	42
<i>nAccelKeys</i>	<i>42</i>
<i>[AccelKey-n]</i>	<i>42</i>
<i>Command</i>	<i>42</i>
<i>Accel</i>	<i>43</i>
Aliases - .ica	44
<i>nAliases=</i>	<i>44</i>
<i>Alias=</i>	<i>45</i>
<i>LocalCommand= and GlobalCommand=</i>	<i>45</i>

3 • Modifying Toolbars & Writing Macros 46

Customizing the Toolbar Look	47
Rearranging Toolbars	47
<i>Dragging & Moving Toolbars</i>	<i>48</i>
<i>Toggling the Display of Toolbars</i>	<i>49</i>
Creating New Toolbars	50
Renaming Toolbars	51
<i>Changing Button Size, Color, and Tooltips</i>	<i>52</i>
Writing Toolbar Macros	53
Simple Macros	54
Intermediate Macros	55
<i>Toolbar Macros Are No Panacea</i>	<i>55</i>
Sharing Toolbars	56
Saving Toolbars	56
Importing Toolbars	56
.mnu File Format	58
General Format	58
Toolbar Format	59
<i>***TOOLBARS</i>	<i>59</i>
<i>**name</i>	<i>59</i>
<i>TBAR_name</i>	<i>59</i>
<i>_Toolbar</i>	<i>59</i>
<i>"titleBar"</i>	<i>59</i>
<i>defaultPosition</i>	<i>59</i>
<i>defaultVisibility</i>	<i>60</i>
<i>xCoord and yCoord</i>	<i>60</i>
<i>rows</i>	<i>60</i>
Button Format	60
<i>ID_cmdName</i>	<i>60</i>
<i>_Button</i>	<i>60</i>
<i>cmdName</i>	<i>60</i>
<i>smallIcon</i>	<i>60</i>
<i>largeIcon</i>	<i>60</i>
<i>macro</i>	<i>60</i>

Flyout Button Format	61
<i>_Flyout</i>	61
<i>_otherIcon</i>	61
<i>TBAR_name</i>	61
Help String Format	61
*** <i>HELPSTRINGS</i>	61
<i>TBAR_name [name]</i>	61
<i>ID_cmdName</i>	61
<i>[helpString]</i>	61

4 • Customizing Menus 62

Modifying the Menu Bar	63
Examining Menu Names	64
<i>Underline - &</i>	65
<i>Dialog Box - .</i>	65
<i>Tab Separator - \t</i>	65
<i>New... and Ctrl+N</i>	65
Editing Macros	65
<i>Cancel - ^C</i>	66
<i>Transparent - '</i>	66
<i>Internationalize - _</i>	66
<i>Enter - ;</i>	66
<i>Pause - \</i>	67
Editing the Help String	67
Changing Options	67
<i>Experience Level</i>	68
<i>MDI Window</i>	69
<i>ActiveX In-Place Activation</i>	69
<i>Checked-State and Grayed-States Variables</i>	70
<i>Value - &</i>	70
<i>Not - !</i>	70
<i>Context Menu Entity Availability</i>	71
<i>Miscellaneous</i>	71
Adding New Menu Items	72
Deleting Menu Items	73
ICM Menu File Format	74
nMenuItem	74
Name	74
<i>Alt-Shortcut - &</i>	74
<i>Dialog Box - .</i>	75
<i>Right-Justified - \t</i>	75
TearOffName	75
<i>Command</i>	75
<i>Cancel - ^C</i>	76
<i>Internationalize - _</i>	76
<i>Enter - ;</i>	76
<i>Pause - \</i>	76
Visibility	76
<i>Experience Level</i>	77
<i>MDI Window</i>	77
<i>ActiveX In-Place Activation</i>	77
<i>Other</i>	78

HelpString	78
SubLevel	78
AddSpacerBefore	79
EntityVisibility	79
ChekVar	80
GrayVar	80
<i>Value - &</i>	80
<i>Not - !</i>	81

5 • Customizing Linetypes 82

Commands That Affect Linetypes	83
System Variables that Affect Linetypes	83
<i>The Special Case of Polylines</i>	83
Compatibility with AutoCAD	84
Customizing Linetypes	84
progeCAD Explorer	84
<i>Editing the Linetype Definition</i>	86
<i>Deleting Linetype Definitions</i>	87
At the Command Prompt	89
Testing the New Linetype	90
Creating Linetypes with the Text Editor	91
The Linetype Format	92
<i>Line 1: Header</i>	92
<i>Line 2: Data</i>	92
Complex (2D) Linetypes	92
Embedding Text	93
<i>Text - "HW"</i>	94
<i>Text Style - STANDARD</i>	94
<i>Text Scale - S=.2</i>	94
<i>Text Rotation - R=0.0</i>	94
<i>Absolute - A=0.0</i>	94
<i>X and Y Offset - X=-0.1 and Y=-0.1</i>	94
Embedding Shapes	95
<i>Shape Name - SSS</i>	95
<i>Shape File - Itypeshp.shx</i>	95

6 • Making Hatch Patterns 96

Where Do Hatch Patterns Come From?	97
How Hatch Patterns Work	97
Creating Custom Hatch Patterns	99
Hatch Command	99
BHatch Command	100
Understanding the .pat Format	101
Comment and Header Lines	101
<i>Comment - ;</i>	101
<i>Start of Definition - *</i>	101

<i>Pattern Name</i>	101
<i>Description</i>	101
The Hatch Data	102
<i>angle</i>	102
<i>xOrigin and yOrigin</i>	102
<i>xOffset and yOffset</i>	102
<i>dash1</i> ,.....	102
Tips on Creating Pattern Codes	103
Adding Custom Patterns to the Palette	105
Creating a Sample Hatch Pattern	105
Creating the Slide	106

7 • Creating Shapes & Fonts 108

Fonts, Complex Linetypes, and GDT Symbols	109
Fonts	109
Complex Linetypes	109
GDT Symbols	109
About Shape Files	110
Font Compatibility with AutoCAD	110
Using Shapes in Drawings	111
The Shape File Format	112
Header Fields	112
<i>Definition Start - *</i>	112
<i>shapeNumber</i>	112
<i>totalBytes</i>	112
<i>shapeName</i>	113
Definition Lines	113
<i>bytes</i>	113
Vector Codes	113
Instruction Codes	114
<i>End of Shape - 0/000</i>	115
<i>Draw Mode - 1/001</i>	115
<i>2/002: Move Mode -</i>	115
<i>Reduced Scale - 3/003</i>	115
<i>Enlarged Scale - 4/004</i>	115
<i>Save (Push) - 5/005</i>	115
<i>Recall (Pop) - 6/006</i>	116
<i>Subshape - 7/007</i>	116
<i>X,y Distance - 8/008</i>	116
<i>X,y Distances - 9/009</i>	116
<i>Octant Arc - 10/00A</i>	116
<i>Fractional Arc - 11/ 00B</i>	117
<i>Bulge Arc - 12/00C</i>	118
<i>Polyarc - 13/00D</i>	118
<i>Flag Vertical Text Flag - 14/00E</i>	118

8 • Using Script Files 120

What are Scripts?	120
Drawbacks to Scripts	121
Strictly Command-Line Oriented	121
Script Commands and Modifiers	122
Script	122
RScript	122
Resume	122
Delay	122
Special Characters	123
<i>Enter - (space)</i>	123
<i>Comment - ;</i>	123
<i>Transparent - '</i>	123
<i>Pause - Backspace</i>	123
<i>Stop - esc</i>	123
Recording Scripts	124

9 • Programming LISP 125

The History of LISP in CAD	126
Compatibility between LISP and AutoLISP	126
<i>Additional LISP Functions</i>	126
<i>Different LISP Functions</i>	127
<i>Missing AutoLISP Functions</i>	127
The LISP Programming Language	127
Simple LISP: Adding Two Numbers	127
LISP in Commands	129
Remembering the Result: setq	130
LISP Function Overview	131
Math Functions	131
Geometric Functions	132
<i>Distance Between Two Points</i>	132
<i>The Angle from 0 Degrees</i>	132
<i>The Intersection of Two Lines</i>	132
<i>Entity Snaps</i>	132
Conditional Functions	133
<i>Other Conditionals</i>	133
String and Conversion Functions	133
<i>Joining Strings of Text</i>	134
<i>Converting Between Text and Numbers</i>	134
<i>Other Conversion Functions</i>	134
External Command Functions	135
<i>Command Function Limitation</i>	136
<i>Accessing System Variables</i>	136
GetXXX Functions	136
Selection Set Functions	137
Entity Manipulation Functions	138
Advanced LISP Functions	138

Writing a Simple LISP Program	139
Why Write a Program?	139
<i>The Id Command</i>	139
The Plan of Attack	139
<i>Obtaining the Coordinates</i>	139
Placing the Text	141
Putting It Together	142
Adding to the Simple LISP Program	143
Conquering Feature Bloat	143
<i>Wishlist Item #1: Naming the Program</i>	143
<i>Defining the Function - defun</i>	144
<i>Naming the Function - C:</i>	144
<i>Local and Global Variables - /</i>	144
<i>Wishlist Item #2: Saving the Program</i>	144
<i>Wishlist Item #3: Automatically Loading the Program</i>	145
<i>Wishlist #4: Using Car and Cdr</i>	145
Saving Data to Files	149
The Three Steps	149
<i>Step 1: Open the File</i>	149
<i>Step 2: Write Data to the File</i>	150
<i>Step 3: Close the File</i>	151
Putting It Together	151
<i>Wishlist #5: Layers</i>	152
<i>Wishlist #6: Text Style</i>	152
Tips in Using LISP	152
<i>Tip #1: Use an ASCII Text Editor.</i>	152
<i>Tip #2: Loading LSP Code into progeCAD</i>	153
<i>Tip #3: Toggling System Variables</i>	153
<i>Tip #4: Be Neat and Tidy.</i>	153
<i>Tip #5: UPPER vs. lowercase</i>	154
<i>Tip # 6: Quotation Marks as Quotation Marks</i>	154
<i>Tip #7: Tabs and Quotation Marks</i>	155

10 • Introduction to DCL..... 156

What Dialog Boxes Are Made Of	158
Your First DCL File	159
DCL Programming Structure	161
<i>Start Dialog Box Definition</i>	161
<i>Dialog Box Title</i>	161
<i>OK Button</i>	161
Testing the DCL Code	164
Displaying System Variable Data	165
Adding the Complimentary LISP Code	166
Clustering Text	167
<i>Supplying the Variable Text</i>	168
<i>Leaving Room for Variable Text</i>	169
Fixing the Button Width	169
<i>Centering the Button</i>	170
Testing the Dialog Box	170
<i>Defining the Command</i>	171

Examples of DCL Coding	173
Buttons	173
<i>Making Buttons Work</i>	174
<i>Check Boxes</i>	176
<i>Radio Buttons</i>	178
Clusters	181
<i>Columns and Rows</i>	181
<i>Boxed Row</i>	182
<i>Boxed Row with Label</i>	182
<i>Special Tiles for Radio Buttons</i>	183
Debugging DCL	184
Dcl_Settings	184
DCL Error Messages	184
<i>Dialog has neither an OK nor a CANCEL button</i>	184
<i>Error in dialog file "filename.dcl", line n</i>	184
<i>Dialog too large to fit on screen</i>	184
Additional Resources	185

11 • DCL Reference 186

Tile Reference	189
<i>Name</i>	190
<i>Label</i>	190
<i>Initial Focus</i>	190
<i>Key</i>	191
<i>Exiting Dialog Boxes</i>	191
Button	192
Radio_Button	196
Toggle	198
Image_Button	199
Edit_Box	201
List_Box	203
Popup_List	206
Slider	208
Text	211
Spacer	214
Image	215
Column	217
Row & Boxed_Row	219
OurBase.Dcl	220
LISP Functions for Dialog Boxes	221
Dialog Boxes Displayed by LISP Functions	222
Load_Dialog	223
New_Dialog	223
Start_Dialog	223
Done_Dialog	224
Term_Dialog	224
Unload_Dialog	224
Get_Tile	225
Set_Tile	225

Get_Attr	225
Mode_Tile	225
Action_Tile	226
Client_Data_Tile	226
Start_List	227
Add_List	227
End_List	227
Start_Image	228
Slide_Image	228
Fill_Image	229
Vector_Image	229
DimX_Tile & DimY_Tile	230
End_Image	230
Dialog Boxes Displayed by LISP Functions	231
Alert	231
Help	231
AcadColorDlg	231
Acad_TrueColorDlg	232
InitDia	233

12 • Employing Diesel Expressions 234

Is Diesel a Programming Language?	234
What Diesel Does	235
Brief List of Diesel Functions	235
<i>Numeric Conversion Functions</i>	236
ModeMacro: Displaying Text on the Status Bar	236
Reporting Values of System Variables	236
Debugging Diesel	237
MacroTrace	238
Using Variables	238
Diesel Functions	239
Math Functions	239
+ (<i>Addition</i>)	239
- (<i>Subtraction</i>)	239
* (<i>Multiplication</i>)	239
/ (<i>Division</i>)	239
Logic Functions	240
= (<i>Equal</i>)	240
< (<i>Less than</i>)	240
> (<i>Greater Than</i>)	240
!= (<i>Not Equal</i>)	240
<= (<i>Less Than or Equal</i>)	240
>= (<i>Greater Than or Equal</i>)	241
and (<i>Logical Bitwise AND</i>)	241
eq	241
if	241
or (<i>Logical Bitwise Or</i>)	241
xor (<i>Logical Bitwise Xor</i>)	241

Conversion Functions	242
<i>angtos</i>	242
<i>fix</i>	242
<i>rtos</i>	243
String Functions	243
<i>index</i>	243
<i>nth</i>	244
<i>strlen</i>	244
<i>substr</i>	245
<i>upper</i>	245
System Functions	245
<i>edtime</i>	245
<i>eval</i>	246
<i>getvar</i>	247
Diesel Programming Tips	247
Diesel in Menus and Toolbars	247
Parsing the Name Macro	248
<i>\$(if, ... !.)</i>	248
<i>\$(eq, ... 1)</i> ,	249
<i>\$(getvar,attmode)</i> ,	249
<i>&Normal</i>	249
Parsing Diesel in Macros	249
Bitcode Macros	250
Diesel in AutoLISP	250
Via the Setvar Function	250
Concatenate Two Diesel Strings	250
Via the MenuCmd Function	251

13 • Understanding DXF 252

References	252
Introduction	253
DXF Formats	253
DWG and DXF Content	254
Miscellaneous Comments	255
DXF Format	255
<i>Header Section of DXF Files</i>	256
Object Properties	260
Group Codes	260
HEADER Section	261
<i>Version Numbers</i>	262
CLASSES Section	262
TABLES Section	263
BLOCKS Section	264
ENTITIES Section	265
OBJECTS Section	267
THUMBNAILIMAGE Section	268
EOF	268

Tailoring the Environment of progeCAD

progeCAD allows you to change the way it looks and works. The first few of these chapters concentrate on changing the *look* of progeCAD; later chapters on changing the way it *works*.

There isn't a whole lot to change in its user interface. For example, you cannot change or configure the devices it works with, such as the graphics board or pointing device. (progeCAD uses whatever devices are specified with the Windows operating system.) You can, however, change a fair bit of progeCAD's user interface, as described by this chapter.

(Later chapters of this ebook include information about progeCAD that you won't find elsewhere. There are a number of aspects of progeCAD that do not seem to be documented anywhere, such as command-line switches, macro metacharacters, and export file formats.)

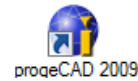
In This Chapter

- Starting progeCAD.
- Command line options.
- Changing screen and cursor colors and displays.
- Setting support file paths.
- Specifying search path options.

Starting progeCAD 2009

You can start progeCAD in one of three ways:

- Double-click the progeCAD icon you found on your computer's desktop.
- On the Windows 2000 or XP taskbars, click the Start button, and then select **Programs | progeCAD 2009 | progeCAD 2009**.
(In Windows Vista and 7, click the Start icon, and then choose **All Programs | progeCAD 2009 | progeCAD 2009**.)
- In the Windows Explorer, double-click the name of a *.dwg* file. (This option works only when progeCAD is assigned to work with *.dwg* files.)



Changing the User Interface

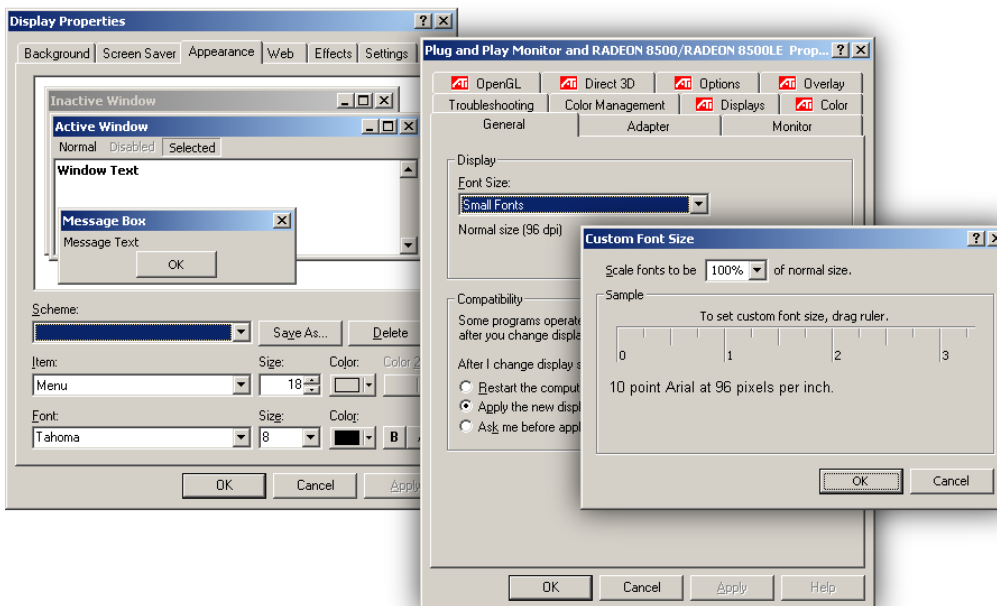
progeCAD allows you to change many aspects of its user interface. You can change the size and color of the crosshair cursor, determine the background color of the drawing area, and more. Some of the changes are controlled by Windows, the others by progeCAD.

Changes Through Windows

Windows lets you set the colors and fonts for windows elements, such as dialog boxes and title bars.

Windows 2000 and XP

To change these in Windows 2000 and XP, right-click the Windows desktop, and select **Properties**.

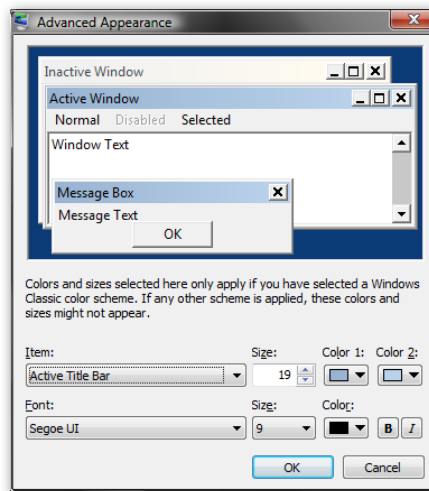


- The **Appearance** tab handles fonts and colors.
- The **Settings | Advanced | General | Display** controls the overall size of fonts and user interface elements.

Windows Vista and 7

In Windows Vista and 7, access is somewhat more complicated:

1. Right-click the desktop, and select **Personalize**.
2. In the Personalization dialog box, choose **Window Color and Appearance**.
3. Choose **Open Classic Appearance Properties for More Color Options**.
4. Click **Advanced**, and then change the color, font, and size of window elements.



To change the font size globally, click **Adjust Font Size** in the Personalization dialog box.

Changes Through progeCAD

Within progeCAD, the Options dialog box controls most of the changes to the user interface, as described in this chapter. Other changes can be made by changing toolbars and menus, as described later in this book.

To access the Options dialog box, enter the **Options** command. (Alternatively, click the Tools menu item, and then click Options.) The dialog box is illustrated on the next page.

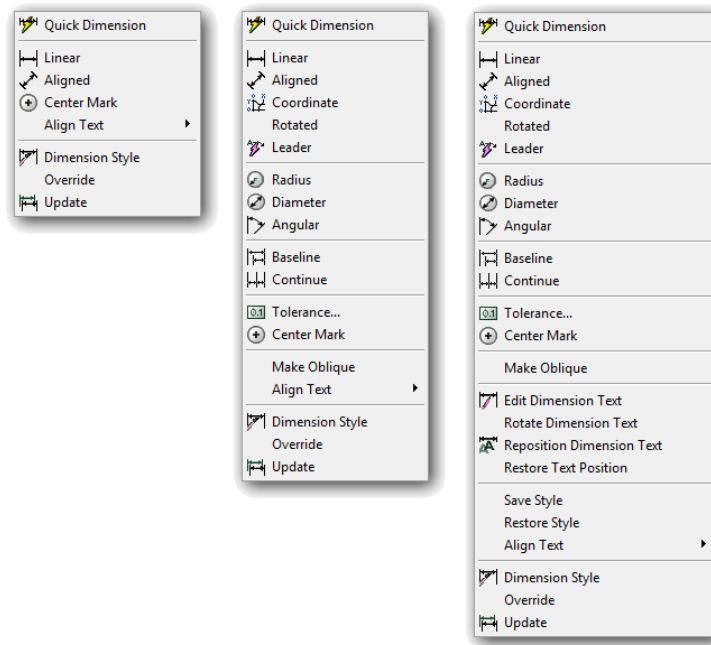
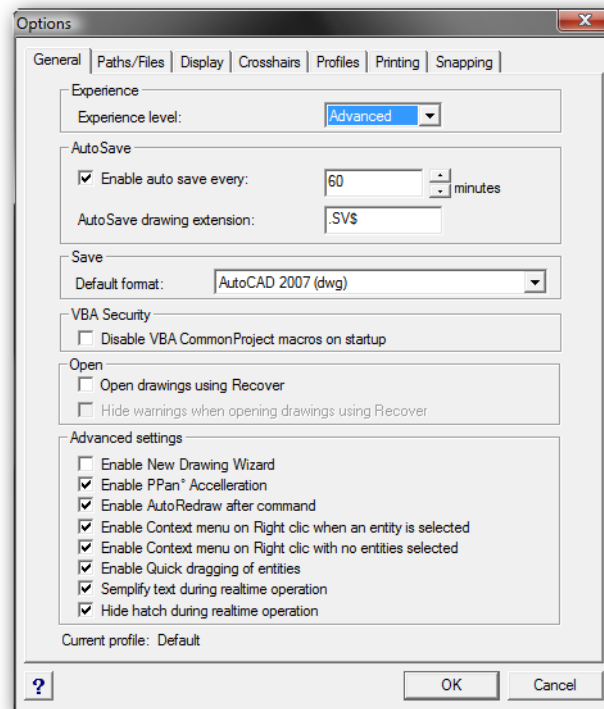
Options: General

Experience Level determines the number of items listed by menus:

- **Advanced** shows all menu items; default.
- **Intermediate** shows many menu items.
- **Beginner** shows the fewest menu items.

Options Dialog Box

The Options dialog box is the primary method of customizing the user interface of progeCAD. You access it with the **Options** command. (From the Tools menu, choose Options.)



Left to right: The Dimension menu in Beginner, Intermediate, and Advanced display modes.

I prefer to keep menus in Advanced mode, but you may find that Beginner mode helps reduce the clutter of commands.

You can customize which menu items are shown by the three experience levels, as described in the chapter on Customizing Menus.

Options: Paths

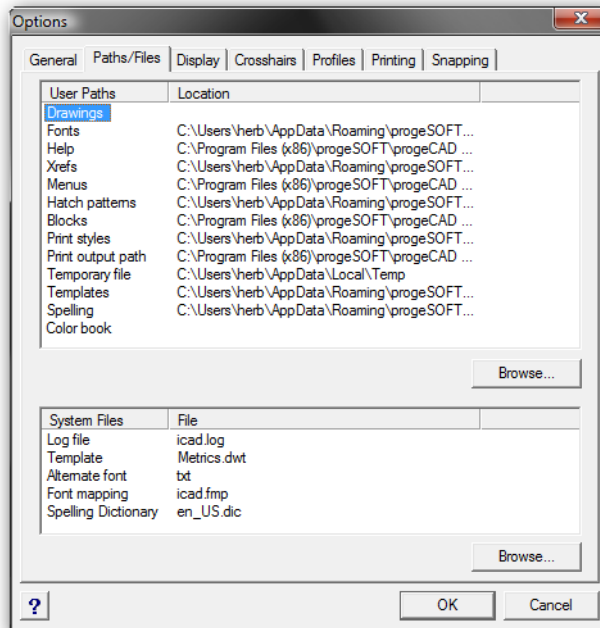
After it is freshly installed on your computer, progeCAD uses a number of folders in which to store support files, such as font files, the on-line help, and hatch patterns. progeCAD finds support files by consulting paths to the folders. (*Paths* specify the name of the drive and the folder, such as *c:\progeCAD*.)

For the most part, you should leave the paths alone. But you may want to change paths for these reasons:

- Your firm has clients with different standards for fonts, layers, and so on. By storing the related files in different folders, and then pointing progeCAD to the folders, you keep the standards separate and intact from each other.
- You are a third-party developer, and need paths pointing to sets of different files.
- You import drawings from other CAD packages, and need to map different sets of fonts via the *.fmp* font mapping file.

Here is how to change paths and files:

1. From the **Tools** menu, select **Options**.
2. In the Options dialog box, select the **Paths/Files** tab.



3. Next to a heading name, click on the path name. For example, next to **Fonts** click

C:\Users\herb\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Fonts...
and then edit the path.

4. To add paths, click the **Browse** button, which lets you select a drive and folder, including those on networks.

When the path name appears missing, such as for Drawings, the default folder is used. For progeCAD, this is *C:\Program Files\progeSOFT\progeCAD 2009*.

TIPS Separate paths with semicolons (;), such as:

D:\CAD\progeCAD 6\Patterns ; D:\CAD\progeCAD 6\Patterns\ISO

When there are two or more paths, progeCAD searches them in the order in which they appear.

In some cases, only the first path is used; for example, if you have two paths for Drawings, the **Open** command looks only in the first one.

Search Path Options

Here are details on the paths specified through the Option dialog box's Paths/Files tab.

Drawings specifies the path to the folders in which commands like Open search for *.dwg* drawing files. Default is *C:\Program Files\progeSOFT\progeCAD 2009*. You may want to change this path to the folder that holds your drawings. I use *C:\Program Files\progeSOFT\progeCAD 2009\samples*, because I often open the sample drawing files.

Fonts specifies the path to folders in which commands such as Style search for *.shx* and *.ps* font files. Defaults are:

- *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Fonts*
- *C:\Program Files (x86)\progeSOFT\progeCAD 2009\Fonts*

progeCAD accesses TrueType fonts automatically through the *C:\Windows\Fonts* folder, so there is no need to add it.

Help specifies the path to the folder holding *.chm* compiled help files used by the Help command. The default is *C:\Program Files\progeSOFT\progeCAD 2009\Help* folder. There is no need to change this path.

Xrefs specifies the paths to the folders in which commands like Xref search for externally-referenced drawing files. Default is *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG*. As with drawings, you should change this to the folder holding your *.dwg* drawing files.

Menus specifies the path to the folders in which commands like Menu search for *.mnu* menu, *.dcl* dialog box, *.lsp* LISP, and *.bmp* bitmap files. Defaults are:

- *C:\Program Files\progeSOFT\progeCAD 2009*
- *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG*
- Location of some bitmap image files used by icons — *C:\Program Files\progeSOFT\progeCAD 2009\bmp*
- Location of addons — *C:\Program Files\progeSOFT\progeCAD 2009\addon*

There is usually no need to change these paths.

Hatch Patterns specifies the path to the folders in which commands like Hatch search for *.pat* hatch pattern and *.sld* slide files. Defaults are:

- Standard hatch patterns — *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Patterns*
- ISO hatch patterns — *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Patterns\ISO*
- Additional hatch patterns — *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Patterns\Extras*

You may want to add paths to *.pat* files provided with other CAD packages, such as AutoCAD.

Blocks specifies the path to the folders in which commands like Insert search for *.dwg* block files. Defaults are:

- *C:\Program Files\progeSOFT\progeCAD 2009*
- *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG*
- *C:\Program Files\progeSOFT\progeCAD 2009\bmp*
- *C:\Program Files\progeSOFT\progeCAD 2009\addon*

You should change this to the folder holding your *.dwg* block files.

Print Styles specifies the path to the folders holding *.ctb* and *.stb* plot style files. The default is *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Plot styles*. You can add paths to plot style files provided with AutoCAD.

Print Output Path specifies the folder in which *.prn* print-to-file spooling files from the Plot command are stored. The default is *C:\Program Files\progeSOFT\progeCAD 2009*. You may need to change this path to work with third-party spooling software.

TIP The **PSetup** command can be used to force plots to be saved as files. In the Print Setup dialog box, click **Spooling**, and then turn on the **Force print to file** option.

Temporary File specifies the path to the folders in which progeCAD stores temporary files, such as automatic backup files (*filename.sv\$*). Default is *C:\Users\login\AppData\Local\Temp*. There is usually no need to change this path.

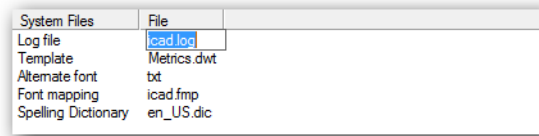
Templates specifies the path to the folders in which commands like New search for *.dwt* template files. Default is *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Templates*.

Spelling specifies the path to the folders in which the Spell command searches for *.dic* dictionary files. Default is *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Spelling*. There is no need to change this path.

Color Books specifies the path to the folders in which commands like Color search for *.acb* color book files. Default is *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\color books*.

Default System File Names

The Option dialog box's Paths/Files tab also allows you to specify the names of default files. These are found in the lower half of the dialog box:



Log File specifies the name of the file used to record the command-line text. (Alternatively, you can specify the name with the `LogFileOn` command.) The default file name is *icad.log*.

TIP You turn on logging with the `LogFileOn` command, and turn it off with `LogFileOff`. You can read the *icad.log* file in any text editor.

Template specifies the name of the drawing file used to start new drawings upon starting progeCAD or when using the New command. Default is *metrics.dwt* in the *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Templates* folder.

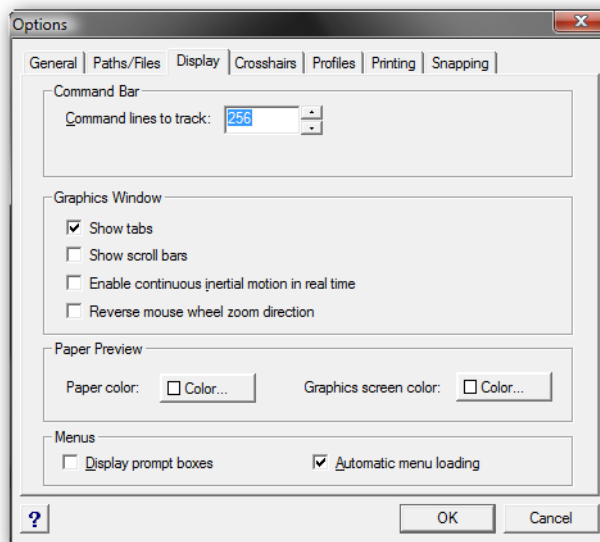
Alternate Font specifies the font to use when a font cannot be found, and has no alternative defined in the *icad.fnt* file. Default is the *txt.shx* font file in the *C:\Program Files\progeSOFT\progeCAD 2009\Fonts* folder. You can also define the alternate font file with the **FontAlt** system variable.

Font Mapping specifies the file that maps fonts. Default is *icad.fmp* in the *\progeCAD 6* folder. When a font cannot be found, progeCAD consults this file for the name of a matched font; if the matched font cannot be found, it uses the alternative font specified above (typically *txt.shx*). For example, progeCAD substitutes its *ic-gdt.shx* font for AutoCAD's *gdt.shx* font.

Spelling Dictionary specifies the default file used by the Spell command to check the spelling of words in drawings. The default is *en_US.dic* found in the *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\Spelling* folder.

Options: Display

The Display tab of the Options dialog box is the most interesting of tabs, as far as customizing the look of progeCAD is concerned.



Graphics Window

Show Tabs toggles the display (shows and hides) the model and layout tabs. I like to have the tabs turned on, because they provide the quickest way to switch between layouts. If you only work in model tab, then you can turn off the tabs.

Show Scroll Bars toggles the display of the vertical and horizontal scroll bars. Again, I like these turned on, because they provide the fastest way to pan the drawing. Alternatively, use the **ScrollBar** command, and then enter **T** to toggle the scroll bars.

```
Command : scrollbar  
WNDLSCRL is currently on: OFF/Toggle/<On>: t
```

Background Colors

The first change I always make is to background color of the drawing area: if it is black, then I change it to white.

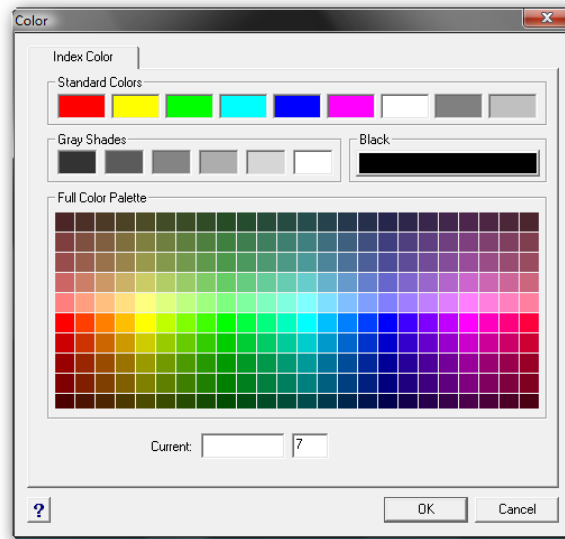
- **Black** was the traditional color in the days when CAD ran on the DOS operating systems; some users prefer it because colors look more vibrant against a black background.
- **White** is preferred by many today, because it most closely resembles the paper upon which the drawing will be printed.

You can change the color of the drawing area in the Graphics Screen Color. To change the colors of the progeCAD drawing area:

1. In the Display tab, click the **Color** button next to Graphics Screen Color.
2. Select a color from the Color dialog box, illustrated below. For a white background, choose color 7.

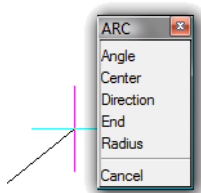
3. Click **OK** to dismiss the Color dialog box.

You can repeat the steps to change the background color of paper space, using the button next to Paper Color.



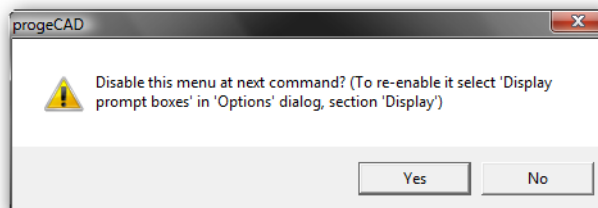
Menus

Display Prompt Boxes toggles the display of context-sensitive menus that appear near the cursor. These menus list the names of options relevant to the current command, as illustrated below for the Arc command.



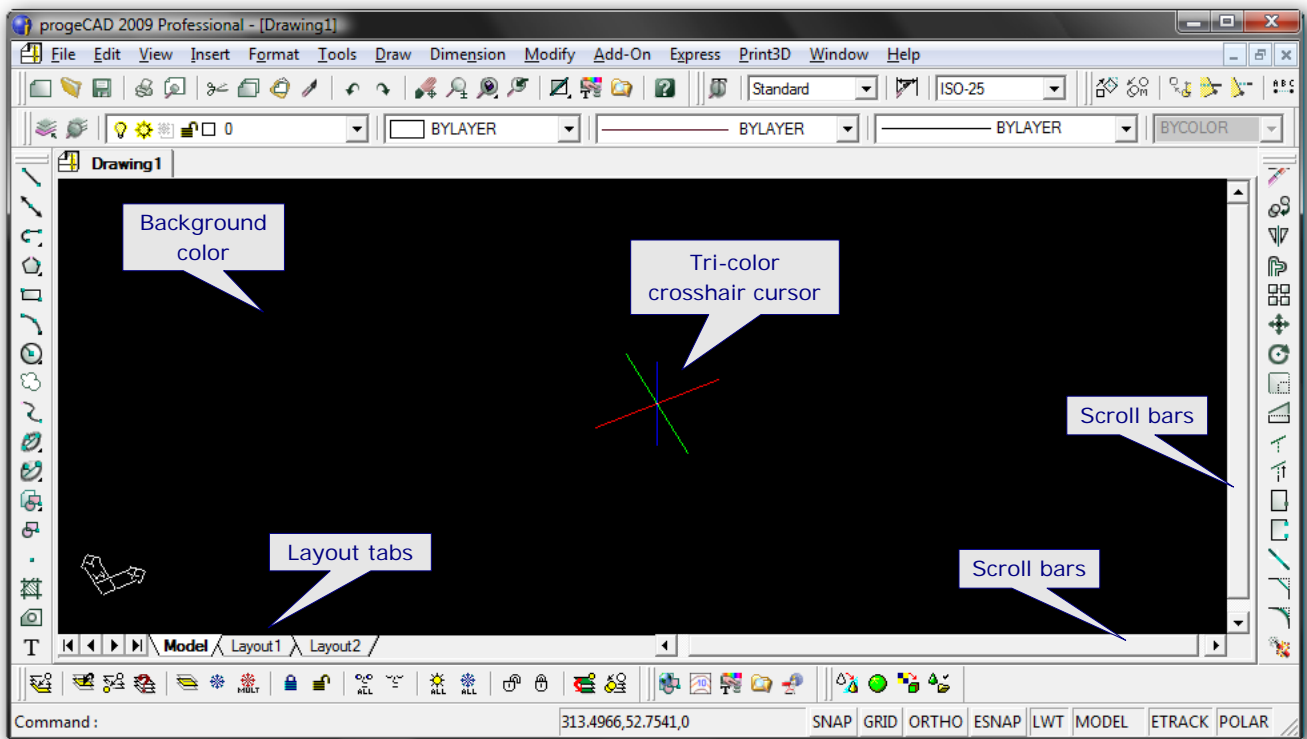
I am not used to using prompt boxes, and so I keep this option turned off.

TIP You can turn off all prompt boxes by clicking the red **x** in the upper right corner. progeCAD asks,



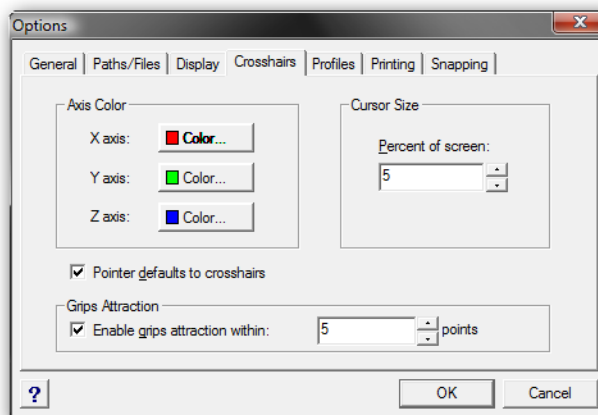
and then click **Yes**.

Clicking **No** turns off the prompt box for this instance of the command only.



Crosshair Colors and Size

The Crosshairs tab of the Options dialog box controls the color and size of the 3D crosshair cursor.



Axis Color

progeCAD displays a “tricolor” cursor that shows a different color for each axis. By default, the colors are:

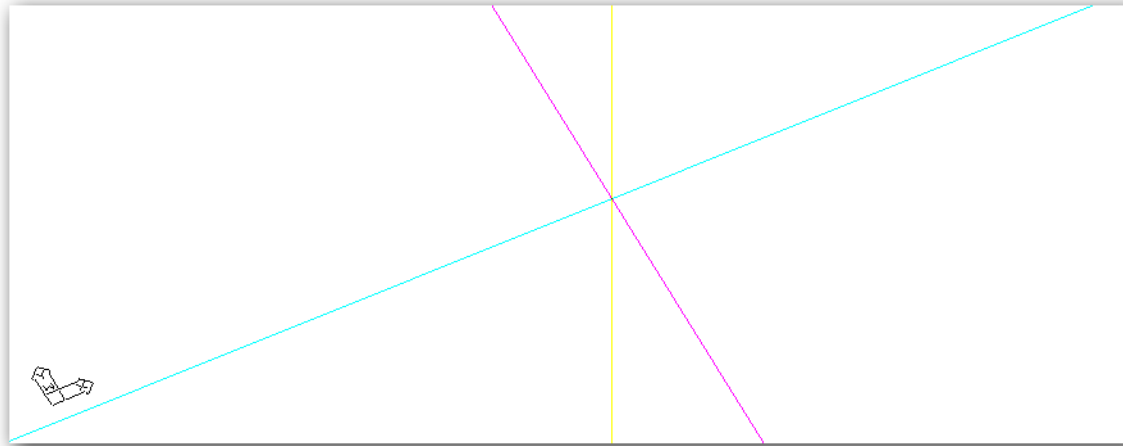
Axis	Color
x	red
y	green
z	blue

But you can change them, if that is your desire. Follow these steps:

1. In the Options dialog box, select the **Crosshairs** tab. Notice the Axis Color section.
2. Click one the three **Color** buttons, and then select another color.
3. Repeat for each of the axes.

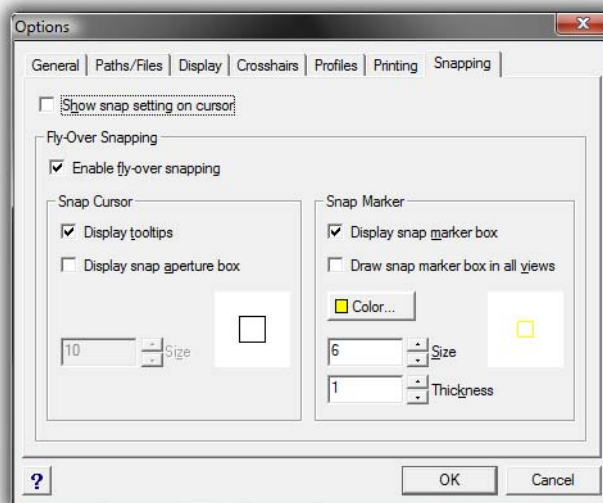
If you wish, you can also change the *size* of the cursor. A setting of 100 percent of the screen makes the cursor full-size, as shown below.

When the background color is other than black, the cursor colors may look different, as shown by the white background illustrated below.

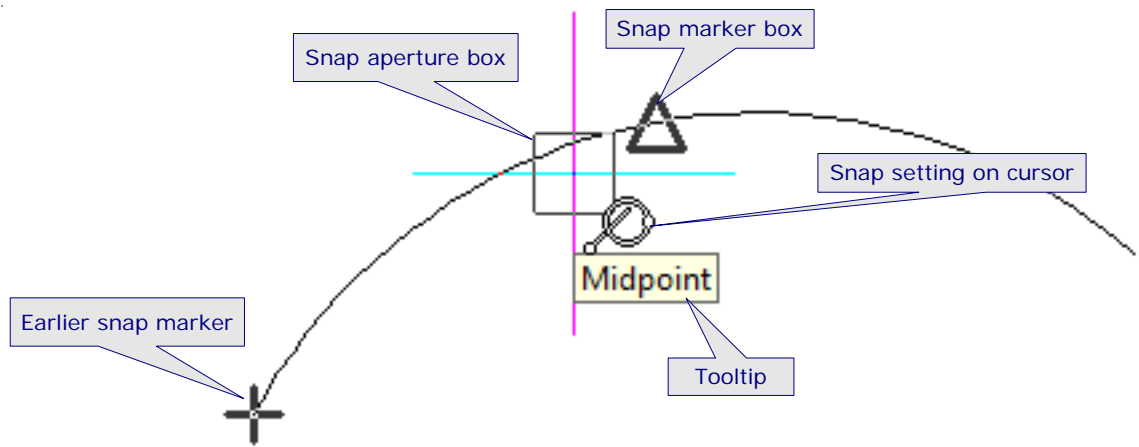



Snap Cursor Colors and Markers

The cursor can display information about entity snaps — called the “flyover” or “snap” cursor. The Snapping tab of the Options dialog box controls the visibility and color of flyovers.



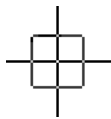
The relationship between the dialog box settings and the flyovers are illustrated below.



 **Show snap setting on cursor** — displays the current entity snap setting near the cursor at all times. Curiously enough, the icons displayed by this setting do not match the icons used by the snap marker box.


Enable fly-over snapping — provides overall control in displaying the tooltips and snap marker box. Although the **Display snap aperture box** option seems to be included, it is unaffected by this toggle.

Quadrant **Display tooltips** — toggles the display of *tooltips*, the small yellow text boxes that name the entity snap. Shown above is the Quadrant tooltip.



Display snap aperture box — toggles the display of the *aperture box*, which defines the area in which progeCAD looks for entities to snap to. In addition, you can change the size of the aperture from its default of 10 pixels.

The tooltips and snap marker appear only when an entity is within the aperture box.

 **Display snap marker box** — toggles the display of the *snap marker box*, which is an icon (not a box!) that indicates the esnap mode. There is a different icon for each entity snap; shown above is the QUADRANT esnap icon. The color, size (in pixels), and thickness can be changed to differentiate it from the aperture box.

Draw snap marker box in all views — shows the snap marker in all viewports, when the drawing is opened in multiple viewports.

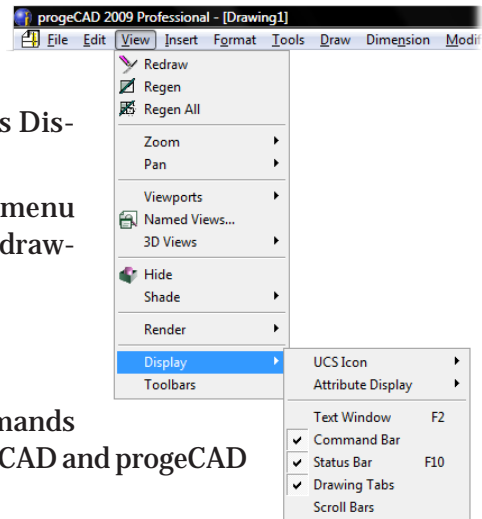
Other User Interface Elements

Other elements of the user interface are changed by commands and by selecting options in the View menu's Display option.

In addition to the scroll bars (described earlier), the menu toggles the display of the command and status bars, and drawing tabs.

Command Bar

The command bar is where you enter the names of commands and respond to prompts from progeCAD. Traditional AutoCAD and progeCAD users prefer to have the command bar visible.



TIP In progeCAD you don't need to keep the command bar displayed, if you want to save some screen real estate, such as with the small screens of netbook computers. When the command bar is off, its function is reproduced on the status bar, as illustrated below.

Using the Circle command in the command bar:



Using the Circle command in the status bar:

Ready	208.0621,9.2473,0	SN
Command : circle	249.6221,4.7903,0	SN
2Point/3Point/RadTanTan/TTT/Arc/Multiple/<Center of circle>:	278.364,73.4516,0	SN
Diameter/<Radius> <59.4161>:	Length: 47.1319	SN

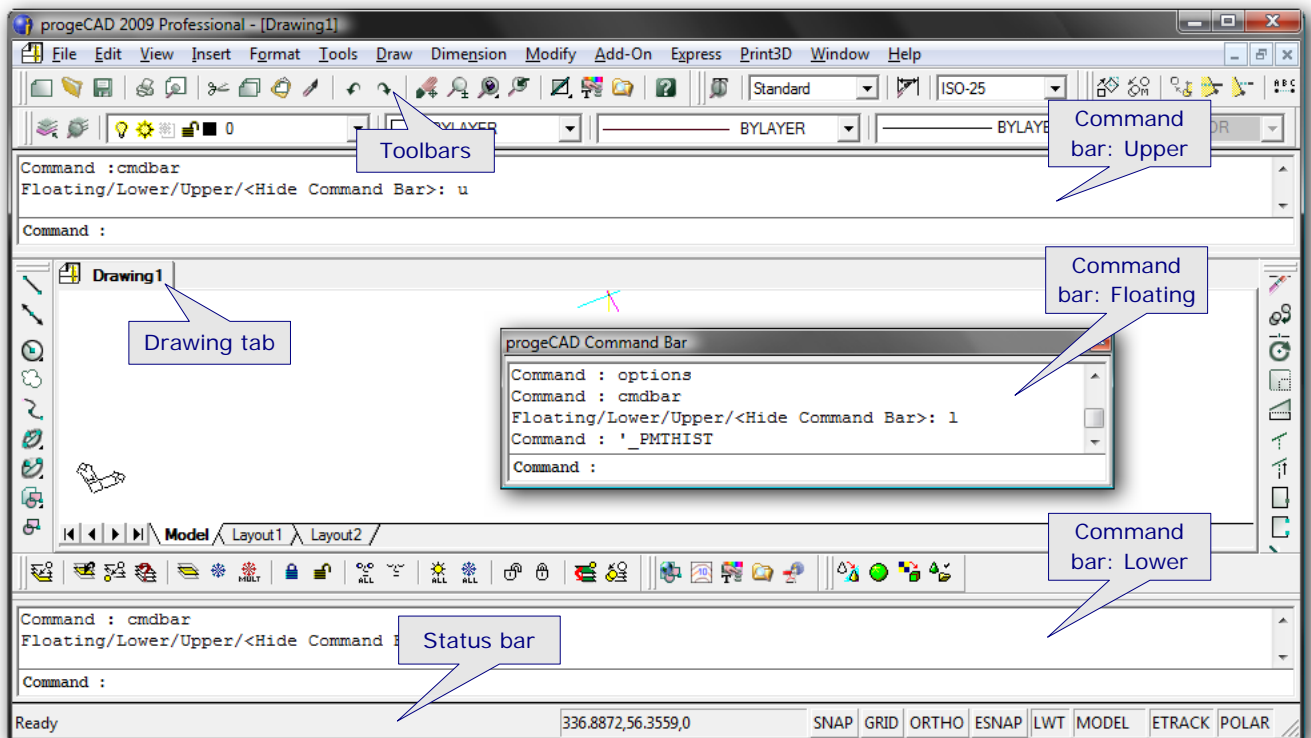
Notice that the coordinate area provides useful information during the command.

From the **View** menu, choose **Display | Command Bar**. But the **CmdBar** command is far more useful:

Command : **cmdbar**
Floating/Lower/Upper/<Hide Command Bar>:

CmdBar	Meaning
Floating	Floats command bar in the drawing area.
Lower	Attaches command bar to top of drawing area (default).
Upper	Attaches command bar to top of drawing area.
Hide	Hides command bar.
T	Toggles display of the command bar (undocumented).

Alternatively, choose **View | Display | Command Bar** from the menu bar (see figure above). Better yet, just click the command prompt area of the status bar to turn the command bar on and off.



Status Bar

I find the status bar one of the most useful elements of a CAD program's user interface, and so that is why I can't fathom why anyone'd want to turn it off. Or why toggling the status bar gets its own dedicated function key — **F10**.

From the **View** menu, choose **Display | Status Bar**. Or press **F10**. Or enter the **StatBar** command:

```
Command : statbar
WNDLSTAT is currently on: OFF/Toggle/<On>: (Enter an option.)
```

StatBar	Meaning
OFF	Hides the status bar.
ON	Displays the status bar.
Toggle	Toggles display of the command bar.

Drawing Tabs

Drawing tabs let you switch quickly between drawings. I am surprised more CAD programs don't have them, since document bars (another name for the tabs) are common in many Windows programs.

TIP To switch between drawings, press **Alt+Tab**.

So, it's cool that progeCAD has the tabs. As with the status bar, I don't know why anyone'd want to turn them off, but you can.

From the **View** menu, choose **Display | Drawing Tabs**. Or enter the **DrawingTab** command:

Command : **drawingtab**

Show/Toggle/<Hide Drawing Tab>: (Enter an option.)

Toolbars

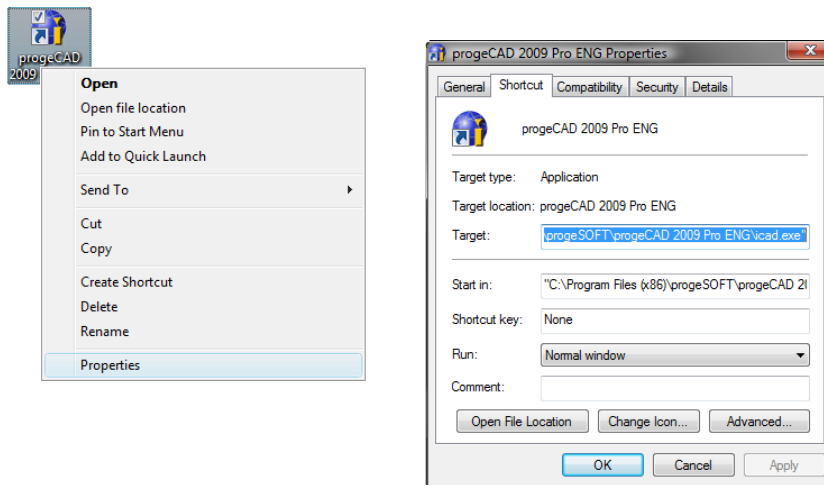
The **Toolbar** item on the View menu displays the Select Toolbars dialog box. It lets you toggle the display of every toolbar, as well as change the size and color of toolbars. (Alternatively, enter the **TbConfig** command.) More details in the “Modifying Toolbars & Writing Macros” chapter.

Startup Options

It was common knowledge in the days of the DOS operating system that many programs had additional options for starting up. With Windows hiding much of what goes on behind a graphical user interface, *command-line options* are no longer in common use. Startup options are still available in some programs, including progeCAD.

For instance, you start progeCAD start with an existing drawing, rather than with a new blank drawing. This is done by editing its *DOS command line*. Here's how:

1. On the Windows desktop, right-click the progeCAD icon.
2. From the shortcut menu, select **Properties**.



3. In the Properties dialog box, select the **Shortcut** tab. Notice that the default command-line text is similar to:

Target: "C:\Program Files\progeSOFT\progeCAD 2009 Pro ENG\icad.exe"

The folder name varies, depending on where progeCAD is located on your computer, and the version of progeCAD you installed. The quotation marks are necessary when the command line text contains spaces. as this one does.

4. Edit the text in the **Target** box to start progeCAD with a drawing. Add the text shown in boldface:

Target: "C:\Program Files\progeSOFT\progeCAD 2009 Pro ENG\icad.exe" **c:\hatch patterns.dwg"**

Notice that:

- The full path to the drawing is required.
- Separate pairs of quotation marks are needed for the program and the drawing file.
- Quotations marks are needed only when the path and file name contain spaces.

5. Click **OK**. to exit the dialog box.

6. Double-click the icon to test your modification. progeCAD should start up with the hatch pattern drawing.

If you make a syntax error, Windows will complain via a dialog box.

Statup Switch

progeCAD also supports the use of switches in startup command lines. A switch tells progeCAD what to do upon startup. Switches are prefixed by the slash characters (/), followed by one or more characters.

/b Switch

The **/b** switch specifies the *.scr* script file to run immediately after progeCAD starts. See the “Using Script Files” chapter to learn how to write script files.

Here is an example of using the **/b** switch to run a script file named “startup script.scr”:

```
Target: "C:\Program Files\progeSOFT\progeCAD 2009 Pro ENG\icad.exe" /b "c:\startup script.scr"
```

There may be additional command-line switches, but I am unaware of them.

Creating Keystroke Shortcuts & Aliases

The best-known ways to execute commands in Windows are with elements of the graphical user interface, such as menus and toolbars. Power users know, however, that the keyboard is the fastest way to enter commands. Once you memorize shortcut keystrokes like **Ctrl+C**, (copy to clipboard), **Ctrl+Tab** (switch to another drawing), and **Ctrl+V** (paste), you can work at top speed.

progeCAD has two facilities for creating shortcut keystrokes of your own. Both are accessed through the Customize dialog box:

- The **Keyboard** tab lets you assign shortcuts to function keys, as well as **Ctrl** and arrow key combinations.
- The **Aliases** tab lets you define *aliases*, which are one- or more-letter command mnemonics, such as **L** for the **Line** command, and **AA** for **Area**.

In This Chapter

- Creating new shortcut keys.
- Editing and deleting keyboard shortcuts.
- Assigning multiple commands.
- Understanding the rules for making aliases.
- Reading the keystroke shortcuts .ick and aliases .ica file formats.
- Making command aliases.
- Editing and deleting aliases.
- Sharing shortcuts with other programs.

Shortcut Keys

Shortcut keys let you carry out commands by simply pressing preassigned keys on the keyboard. For example, pressing **Ctrl+S** to save drawings. For some users, this is faster than selecting commands from menus, toolbars, or typing entire command words at the keyboard.

Out-of-the-box, progeCAD defines the shortcut keystrokes shown below.

Shortcut	Meaning	Command Executed
Function Keys		
F1 ^a [Ⓜ]	Display on-line help.	HELP
F2 ^a	Display Prompt History window.	PMTHIST
F3 ^a	Toggle object snap mode.	ESNAPT
F4 ^a	Toggle tablet mode.	TABLET T
F5 ^a	Switch to next isoplane.	ISOPLANE
F6 ^a	Toggle coordinate display.	COORDINATET
F7 ^a	Toggle display of the grid.	GRID T
F8 ^a	Toggle ortho mode.	ORTHOGONAL T
F9 ^a	Toggle snap mode.	SNAPT
F10	Toggle status bar.	STATBAR T
Control Keys		
Ctrl+A ^a [Ⓜ]	Select all entities in drawing.	SELGRIPS ALL
Ctrl+C ^a [Ⓜ]	Copy selected entities to Clipboard.	COPYCLIP
Ctrl+Shift+C	Copies entities with a basepoint.	COPYBASE
Ctrl+D	Copies large numbers of entities.	COPYQUICK
Ctrl+Shift+D	Copies with basepoint.	COPYBASEQUICK
Ctrl+E ^a	Switch to next isoplane.	ISOPLANE
Ctrl+N ^a [Ⓜ]	Start a new drawing.	NEW
Ctrl+O ^a [Ⓜ]	Open a drawing file.	OPEN
Ctrl+P ^a [Ⓜ]	Print the drawing.	PRINT
Ctrl+S ^a [Ⓜ]	Save the drawing.	QSAVE
Ctrl+T ^a	Toggle tablet mode.	TABLET T
Ctrl+V ^a [Ⓜ]	Paste from Clipboard into drawing.	PASTECLIP
Ctrl+X ^a [Ⓜ]	Cut selected entities to Clipboard.	CUTCLIP
Ctrl+Y ^a [Ⓜ]	Redo the last undo.	REDO
Ctrl+Z ^a [Ⓜ]	Undo the last command.	U
Ctrl+I ^a	Toggles the Properties palette.	PROPERTIES
Ctrl+2 ^a	Displays the Explorer dialog box.	EXPLAYERS
Other Keys		
Del ^a [Ⓜ]	Deletes selected entities .	DELETE
Down Arrow	Scrolls recent commands.	...
Up Arrow	Scrolls recent commands.	...
Shift+Left Arrow	Pans left.	PAN L
Shift+Right Arrow	Pans right.	PAN R
Shift+Down Arrow	Pans down.	PAN D
Shift+Up Arrow	Pans up.	PAN U
Page Down	Pans down by a pagefull.	PAN PGD
Page Up	Pans up by a pagefull.	PAN PGU

^a Shortcut key compatible with AutoCAD.

[Ⓜ] Shortcut key standard in Windows.

Shortcut keys can be assigned to:

- **Function keys** — those marked with the **F** prefix, such as **F1** and **F2**.
- **Shifted keys** — hold down the **SHIFT** key, and then press a function, number, or alphabet key, such as **F2** or **B**.
- **Alternate keys** — hold down the **Alt** key, and then press another key.
- **Control keys** — hold down the **CTRL** key, and press another key.
- **Shifted Control keys** — hold down the **SHIFT** and **CTRL** keys, and press another key.
- **Shifted Alternate keys** — hold down both the **SHIFT** and **ALT** keys, and then press another key.
- **Control + Alternate keys** — hold down both the **CTRL** and **ALT** keys, and press another key.
- **Shifted Control + Alternate keys** — hold down the **CTRL** and **ALT** and **SHIFT** keys, and press another key.

TIP It does not matter if you press **SHIFT** first or **CTRL** first — similarly for **ALT**.

You can, of course, add and change definitions, assigning commands to as many as 188 key combinations.

TIP You should *not* change keys reserved by Windows, especially these:

F1	Display help.
CTRL+F4	Close the current window.
ALT+F4	Exit progeCAD.
CTRL+F6	Change focus to the next window.

Defining Shortcut Keys

Here are the steps to define shortcut keys. In this tutorial, you assign the **Fillet** command to **CTRL+SHIFT+F**:

1. From the menu bar, select **Tools | Customize | Menu**. Select the **Keyboard** tab.

The shortcuts currently assigned are listed in the left-hand column. All of progeCAD's commands are listed in alphabetical order in the right-hand column.

2. Click **New**. Notice that text fields become available (ungray themselves) at the bottom of the dialog box.

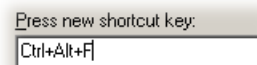
3. In the **Press new shortcut key** text entry box, press the key combination:

Press new shortcut key: (Press and hold the **Ctrl** key.)

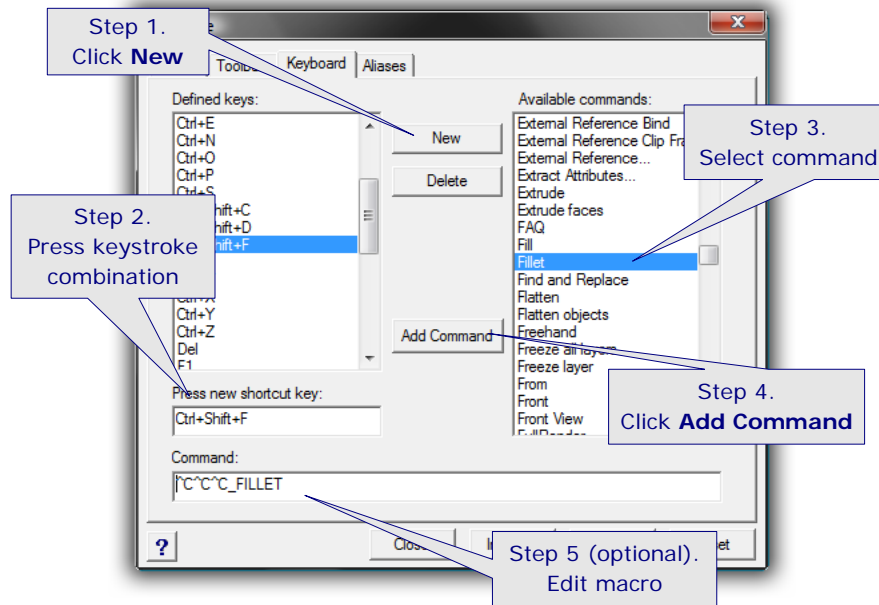
(Hold down the **Alt** key.)

(Press **F**, and then let go.)

Notice that **Ctrl+Alt+F** is added to the list of Defined Keys.



Press new shortcut key:
Ctrl+Alt+F



4. From the **Available Commands** list, select **Fillet**, the command you wish to assign to a keystroke.
5. Click **Add Command**. Notice that `^C^C^C_FILLET` appears in the **Command** box.

There are three `^c`s prefixing the **Fillet** command. They are short for “cancel.” (`^c` is called a special character, or a *metacharacter*. You learn more about them in the “Modifying Toolbars & Writing Macros” chapter.)

The underscore (`_`) “internationalizes” the command, so that the English version works in progeCAD localized for other languages. This ensures the macro works on all dialects of the CAD software.

6. Click **Close** to dismiss the dialog box.
7. Test the keystroke shortcut: hold down the **CTRL** and **SHIFT** keys, and then press **F**. progeCAD should execute the **Fillet** command.

TIPS You can assign one or more keystroke shortcuts to commands.

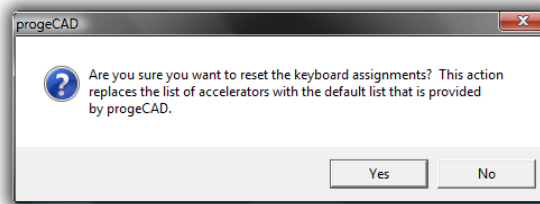
Once a keystroke is assigned, it cannot be used with other commands.

Editing Keyboard Shortcuts

To edit keyboard shortcuts, follow these steps:

1. In the Customize dialog box’s Keyboard tab, select a keystroke under **Defined Keys**, such as the **CTRL+SHIFT+F** we defined earlier.
2. Click the **Command** area, and then edit the command. You can backspace over the command text to erase it, and then select another command.
3. Click **Close** to exit the dialog box.

TIP If you wish to return progeCAD's keyboard shortcuts to their original configuration, click the **Reset** button. progeCAD displays a warning dialog box:



Click **Yes**, if you are sure. If you are not sure, click **No**, and then export the definitions using the **Export** button before resetting the definitions.

Deleting Keyboard Shortcuts

To remove a keyboard shortcut, follow these steps:

1. In the Keyboard tab of the Customize dialog box, select a keystroke under **Defined Keys**, such as the **CTRL+SHIFT+F** you edited above.
2. Click **Delete**. The shortcut is removed without warning.
3. Click **Close** to exit the dialog box.

Assigning Multiple Commands

You can assign more than one command to a keyboard shortcut. When two or more commands are executed together, they are called a *macro*. (You learn more about macros in the “Modify-ing Toolbars & Writing Macros” chapter.)

For example, you can copy all objects in the drawing to the Clipboard with two commands: **Select All** to select all objects, followed by **CopyClip** to copy them to the Clipboard. These two can be combined into a single keystroke, as follows:

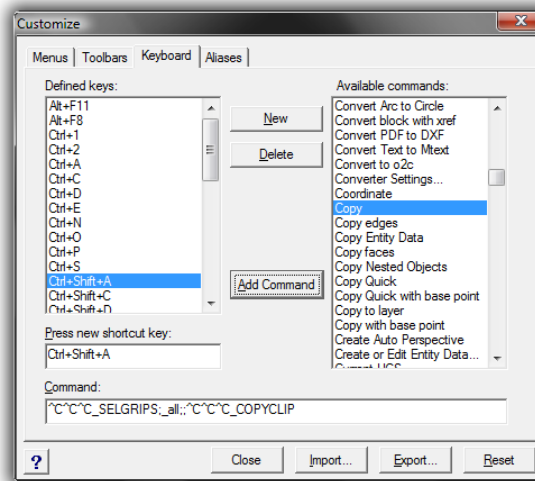
1. In the Keyboard tab of the Customize dialog box, click **New**.
2. Press **CTRL+SHIFT+A**.
3. Under the Available Commands list, select **Select All**, and then click **Add Command**.

TIP The quick way to find a command in the I-o-n-g **Available Commands** list is to type the first letter of the command. For example, type **s** to get to the commands starting with “S.”

4. Now select **Copy** from Available Commands, and then click **Add Command**. Notice that macros for both commands appear in the **Command** box:

```
^C^C^C_SELGRIPS;_all;;^C^C^C_COPYCLIP
```

The semicolon (;) is another metacharacter. It is equivalent to pressing **Enter**.



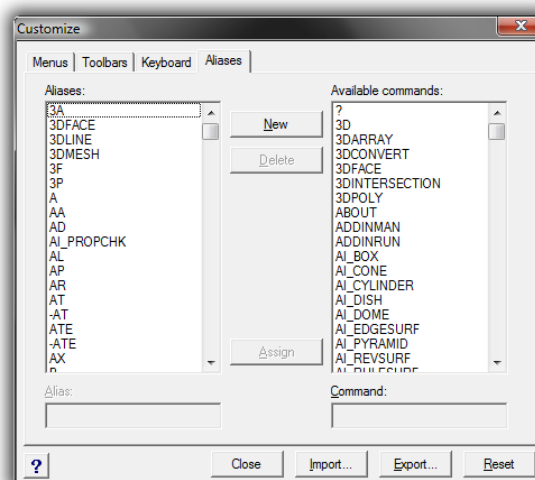
5. Click **Close**.
6. Test the macro by pressing **CTRL+SHIFT+A**. progeCAD reports:
 - : _SELGRIPS
 - Select entities to display grips: _ALL
 - Select entities to display grips:
 - : _COPYCLIP

Try pasting the copied objects into another document using **CTRL+V**.

Command Aliases

In addition to keystroke shortcuts, progeCAD also allows you to define one- and multi-letter command shortcuts called *aliases*. Aliases are abbreviations of command names, such as **L** for the Line command and **CP** for Copy.

progeCAD predefines 245 aliases, and you can see the list of them by following these steps.

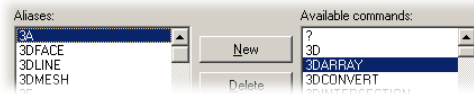


1. From the **Tools** menu, select **Customize | Menu**. (Or, enter the **Customize** command in the command bar.) Notice the Customize dialog box.

2. In the Customize dialog box, choose the **Aliases** tab.

In the Aliases column, you have the list of names of aliases already defined by the progeCAD programming team. The Available Commands column lists the names of all the commands found in progeCAD. (Not all commands have an alias assigned to them.)

3. Let's look at how aliases are linked to commands. Under the Aliases column, click an alias, such as **3A**. Notice that it is linked with the **3DArray** command in the Available Commands column.



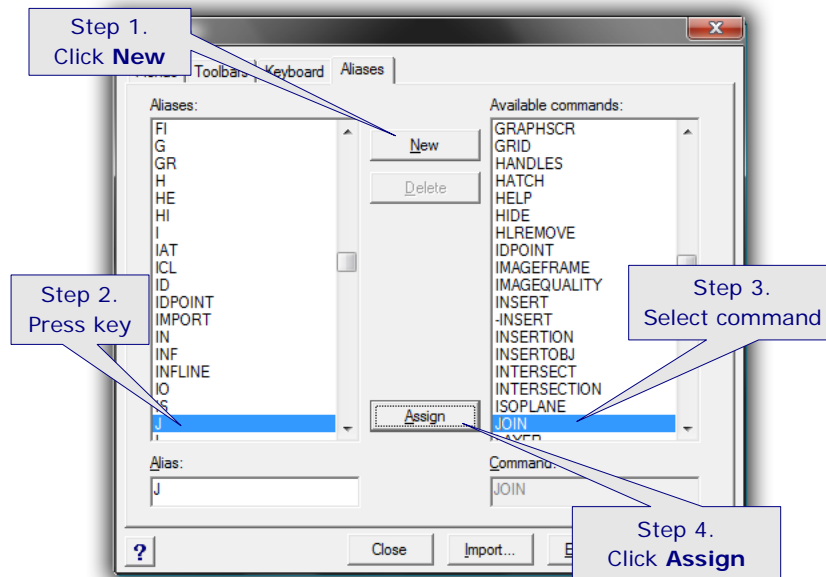
Aliases can be any length, but when they contain more than two or three characters, then they start to defeat the purpose of aliases — being shortcuts to command names!

Still, there is one reason for having long alias names: it makes progeCAD commands compatible with AutoCAD. For instance, 3DFace is an AutoCAD command, and the alias for the progeCAD's Face command. (Many of IntelliCAD's commands have names identical to AutoCAD's, but it's not clear to me why some are different. Aliases fix the problem.)

Creating New Aliases

Creating and editing aliases in progeCAD is simpler than in AutoCAD, where you have to edit its *acad.pgp* file with a text editor. In progeCAD, you work with the Customize dialog box — much easier!

For this tutorial, we create an alias “J” for the **Join** command.



1. In the Customize dialog box's **Aliases** tab, click New.
2. In the Alias text entry box, enter **J**.
3. In the Available Commands list, select **Join**,

4. Click **Assign**.
5. Click **Close** to dismiss the dialog box.
6. Test the alias by entering **J** and then pressing **Enter**. progeCAD should execute the **Join** command.

TIPS Aliases can be used in toolbar and menu macros.

Unlike keyboard shortcuts, aliases cannot be macros. Aliases consist of a single command only; no options, no multiple commands, no metacharacters.

Editing Aliases

To edit an alias, follow these steps:

1. In the **Alias** text entry box, enter the alias you wish to edit, such as the **J** we defined above.
2. In the Available Commands list, select another command, and then click **Assign**.

Deleting Aliases

To remove an alias, follow these steps:

1. In the Alias text entry box, enter the alias you wish to delete, such as the **J** we used earlier.
2. Click **Delete**. Notice that the alias is removed from the Aliases list.
3. Click **Close** to exit the dialog box.

Rules for Writing Aliases

Here are some suggestions Autodesk provides for creating command aliases:

- An alias should reduce a command to two characters at most.
- Commands with a control-key equivalent, status bar button, or function key do not require a command alias. Examples of commands to avoid include the **New** command (already assigned to **Ctrl+N**), **Snap** (already on the status line), and **Help** (already assigned to function key **F1**).
- Try to assign the first character of a command. If it is already taken by another command, assign the first two, and so on. For example, **C** is assigned to the **Circle** command, while **CP** is assigned to the **Copy** command.
- For consistency, add suffixes for related aliases. For example, **H** is assigned to the **Hatch** command, so assign **HE** for **HatchEdit**.

progeCAD Aliases: Sorted by Command Name

A

Align	Al
Aperture	Ap
Arc	A
Area	Aa
Array	Ar
AttDef	-At
AttDisp	Ad
AttEdit	-Ate
AttExt	Ax

B

Backgrounds	Background
Base	Ba
Blipmode	Bm
Block	-B
Boundary	Bo
	Bpoly
-Boundary	-Bo
Break	Br

C

Chamfer	Cha
Change	-Ch
Circle	C
Config	Options
	Pr
	Preferences
	Rconfig
Copy	Co
	Cp
Copylink	Cl

D

DdAttDef	At
DdAttE	Ate
DdChProp	Ch
DdEdit	Ed
DdGrips	Gr
DdInsert	I
DdRename	Ren
DdrModes	Rm
DdSelect	Se
DdUnits	Un
Dist	Di
Divide	Div
Donut	Bagel
	Do
DText	Dt
DView	Dv
DxfIn	Dn
DxfOut	Dx

DimenSions

DimAligned	Dal
	Dimali
DimAngular	Dan
	Dimang
DimBaseline	Db
	Dbase
DimCenter	Dce
DimContinue	Dco
	Dimcont
DimDiameter	Ddi
	Dimdia
DimEdit	Ded
	Dimed
Dimension	Dim
DimLinear	Dli
	Dimlin
DimOrdinate	Dor
	Dimord
DimOverride	Dov
	Dimover
DimRadius	Dra
	Dimrad
DimStyle	Dst
	Dimsty
DimTEdit	Dimted

E	
EditLen	Len
	Lengthen
EditPline	Pe
	Pedit
Ellipse	El
EntProp	Ai_Propchk
	Ddmodify
	Mo
	Mtprop
Erase	E
ESnap	-Os
	-Osnap
	Osnap
ExpBlocks	B
	Bmake
	Xb
	Xr
ExpDimStyles	Ds
ExpFonts	Ddstyle,
	Expstyle
	Expstyles
ExpLayers	Ddlmodes
	Explorer
	La
Explode	X
ExpLtypes	Ddltype
	Lt

Export	Exp
ExpUcs	Dducs
	Uc
ExpViews	Ddview
	V
Extend	Ex
Extrude	Ext

F

Face	3dface
	3f
Fillet	F
Filter	Fi
Font	St
	Style
Freehand	Sketch

G

Grid	G
------	---

H

Hatch	H
HatchEdit	He
Hide	Hi

I

IdPoint	Id
ImageAdjust	Iad
ImageAttach	Iat
ImageClip	Icl
InfLine	Xl
	Xline
InsertObj	Io
Interfere	Inf
Intersect	In
Isoplane	Is

L

-Layer	-La
Leader	Le
	Lead
Lighting	Light
Line	3dline
	L
	Li
-Linetype	-Lt
List	Ls
LtScale	Lts

M

MatchProp	Ma
Materials	Rmat
Mesh	3dmesh
Mirror	Mi
MLine	DI
	Dline
Move	M
MSnapShot	Mslide
MSpace	Ms
MText	Mt
	T
MView	Mv

N

New	N
NewWiz	Ddnew

O

Open	Imp
	Import
	Op
OpenImage	Image
Orthogonal	Or
	Ortho

P

-Pan	-P
	P
Parallel	O
	Offset
PasteSpec	Pa
Plane	So
	Solid
Point	Po
Polygon	Pol
Polyline	Pl
	Pline
PPreview	Makepreview
	Pre
	Preview
PSpace	Ps
Purge	Pu

Q

QText	Qt
-------	----

R

Rectangle	Rec
	Rect
	Rectang
	R
Redraw	Ra
RedrawAll	Re
Regen	Rea
RegenAll	Reg
Region	Reg
ReInit	Ri
Rename	-Ren
Render	Rr
Revolve	Rev
Rotate	Ro

S

Save	Sa
Scale	Sc
Script	Scr
Section	Sec
SelGrips	Selgrip
SetColor	Col
	Ddcolor
SetDim	D
	Ddim
SetESnap	Ddesnap
	Ddosnap
	Os
SetRender	Rpref
SetUcs	Dducsp
	Ucp
SetVar	Set
Shade	Sha
Slice	Sl
Snap	Sn
Spell	Sp
Spline	Spl
SplinEdit	Spe
Stretch	S
Subtract	Su

T

Tablet	Ta
TbConfig	To
Text	-T
	Tx
Time	Ti
Tolerance	Tol
Torus	Tor
Trim	Tr

U

Undelete	Oo
	Oops
Union	Uni
Units	-Un

V

Vba	Vbaide
View	-V
ViewCtl	Ddvpoin
	Vp
Viewpoint	-Vp
	Vpoint
VpLayer	VI
VPorts	Vport
	Vw
VSnapshot	Vs
	Vslide

W

WBlock	W
WCloseAll	Closeall
Wedge	We
WmfIn	Wi
WmfOut	Wo

X

XBind	-Xb
XClip	Clip
-Xref	-Xr
	Xa
	Xattach

Z

Zoom	Z
------	---

3

3dArray	3a
3dPoly	3p

Sharing Shortcuts

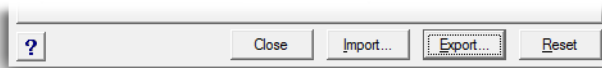
progeCAD makes it easy to share keyboard shortcuts and aliases with other IntelliCAD and AutoCAD users. (While AutoCAD can share its aliases with progeCAD, it cannot share its keyboard shortcuts.)

For example, a CAD manager writes a standard set of shortcuts, and then wants to install them on all other computers running progeCAD. Here's what the manager needs to do: (1) export the shortcuts from his copy of progeCAD; and then (2) import the shortcuts to the copies of progeCAD running on other computers. The sharing can be done with a USB key or over the network.

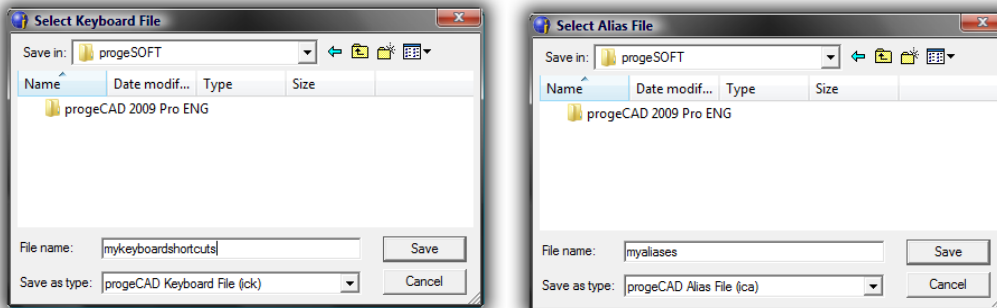
Exporting Shortcuts & Aliases

Here are the steps to exporting shortcuts from progeCAD:

1. From the **Tools** menu bar, select **Customize | Menu**.
 - To export keyboard shortcuts, choose the **Keyboard** tab.
 - To export aliases, choose the **Aliases** tab.
2. Click the **Export** button.



3. Enter a file name in the dialog box. (Ignore the dialog box's incorrect title; you don't "select file" in this step.) If necessary, change the folder name.



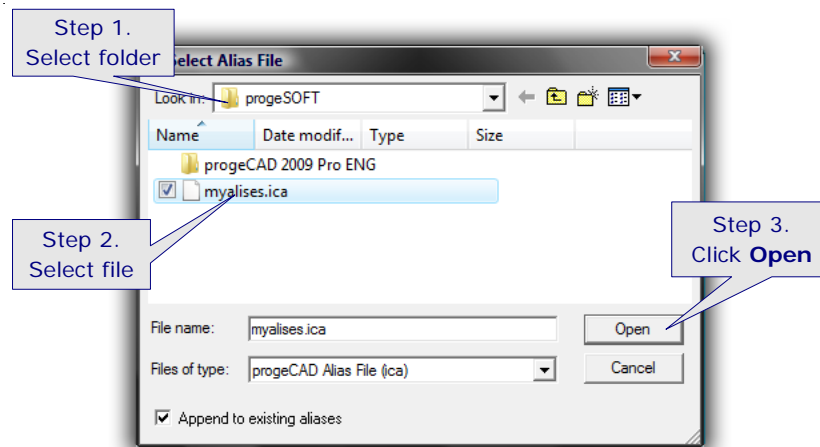
- Keyboard shortcuts can only be saved in ICK (IntelliCAD keyboard, not "icky") format.
 - Aliases can be saved in ICA (IntelliCAD Aliases) or PGP (program parameters) formats. To share aliases with AutoCAD users, select "AutoCAD Alias File (pgp)" from the **Save as type** drop list.
4. Click **Save**.

Importing Shortcuts and Aliases

With the keyboard and alias definitions stored in *.ick* and *.ica* files, you can copy the files to other computers through the network, on USB plug, or via an email attachment.

To import the files into progeCAD, follow these steps:

1. From the **Tools** menu bar, select **Customize | Menu**.
 - To import keyboard shortcuts, choose the **Keyboard** tab.
 - To import aliases, choose the **Aliases** tab.
2. Click **Import**, and select the file name in the dialog box. (If necessary, change the folder name.)



3. *For aliases only:*

Files of type: decide whether you want to import *.ica* aliases or AutoCAD *.pgp* aliases.

Append to existing aliases: decide whether you want existing aliases overwritten with the new ones, or if the new ones should be appended.

TIP If you used the **Append to existing aliases** option, you may, unfortunately, notice double definitions.

If this causes a problem, use the Customize dialog box's **Reset** button to clear the duplicates, and then repeat the **Import** command without the **Append** option turned on.

4. Click **Open**. Notice the newly added shortcuts and/or aliases.

Importing Through the Command Bar

A pair of commands allow you to import shortcuts at the command-line:

ReadAccelerators prompts to open an *.ick* file to read keyboard accelerators.

Accelerator file name: (Enter path and name of *.ick* file.)

ReadAliases prompts to open an *.ica* file to read keyboard accelerators.

Alias file name: (Enter path and name of *.ica* file.)

File Formats

As shown in the previous section, the **Export** button of the Customize dialog box saves keystroke shortcuts and aliases to files. In this section, we look at the formats of these files. Note that “accelerator” is another term for “keyboard shortcut.”

Keystroke Shortcuts - .ick

After exporting keystroke shortcuts from progeCAD, the first few lines of the .ick file look like this:

```
[progeCAD Custom Keyboard File]
nAccelKeys=34
[AccelKey-0]
Command=^C^C^C_NEW
accel=9,78,19600
[AccelKey-1]
Command=^C^C^C_OPEN
accel=9,79,19601
```

Shortcuts that you create are added at the end of the file. progeCAD does not document the .ick file, so here is an overview of the meaning of its contents.

TIP progeCAD *hardwires* its default keystroke shortcuts. This means that they are part of the program code, and so this feature makes it easy to reset shortcuts to their original condition.

nAccelKeys

The **nAccelKeys** item starts off the file (short for “number of accelerator keys”). It is the total number of shortcuts in the .ick file. This value is generated by progeCAD.

```
nAccelKeys=34
```

[AccelKey-n]

The **AccelKey-** item indicates the start of the next accelerator key definition. progeCAD numbers them sequentially. Items in square brackets are comments.

```
[AccelKey-0]
```

Command

The **Command** item defines the macro executed by the shortcut. The macro uses the same syntax and metacharacters as toolbar and menu macros. See Chapter 4 for details.

```
Command=^C^C^C_NEW
```

Accel

The **Accel** item defines the keystroke that is the shortcut (short for “accelerator”).

```
accel=9,78,19600
```

progeCAD uses three code numbers to define accelerators:

- **9** — first number is a code for the prefix key, such as **SHIFT** and **CTRL**.
- **78** — second number is the code for key, such as **A** and **F1**.
- **19600** — third number appears to define the source of the shortcut.

Prefix keys have the following values:

Value	Meaning
1	Unshifted
5	SHIFT
9	CTRL
13	CTRL + SHIFT
17	ALT
21	ALT + SHIFT
25	ALT + CTRL
29	ALT + CTRL + SHIFT

The letters A through Z have the same values as their ASCII codes, 65 (A) through 90 (Z). Numerals also use ASCII codes, 48 (0) through 57 (9). Keys have the following values:

Value	Meaning
Alpha-numeric Keys	
48	Number 0
...	...
57	Number 9
65	Letter A
...	...
90	Letter Z
Function Keys	
112	F1
113	F2
114	F3
115	F4
116	F5
117	F6
118	F7
119	F8
120	F9
121	F10
122	F11
123	F12

Value	Meaning
Cursor Keys	
33	PgUp
34	PgDn
35	End
36	Home
37	Left
38	Up
39	Right
40	Down
45	Insert
46	Delete
Numeric Keypad	
106	* (<i>star</i>)
107	+ (<i>plus</i>)
109	- (<i>minus</i>)
111	/ (<i>divide</i>)

To define shortcuts, you can use any combination of prefix keys (such as **CTRL** and **ALT+SHIFT**) and another key, such as **9** or **HOME**.

For example, **CTRL+9** would be 9, 57 while **ALT+SHIFT+HOME** is 21,36.

I am unable to determine the meaning of the third number to my satisfaction. For default shortcuts, the number increments from 19600 to 19633. When I create shortcuts, however, the number is always 0. My guess is that the third number identifies the source of the shortcut: an IntelliCAD default (19600-range) or user-created (0).

Value	Meaning
0	User-defined shortcut
19600	Default shortcut.

Aliases - .ica

After exporting aliases from progeCAD, the first few lines of the *.ica* file look like this:

```
[progeCAD Custom Alias File]
nAliases=255
[Alias-0]
Alias=3A
LocalCommand=3DARRAY
GlobalCommand=_3DARRAY
```

progeCAD does not document the *.ica* file, but it shares some similarities with the *.ika* file. Here is an overview of the meaning of its entries.

nAliases=

The **nAliases=** item starts off the file, and is the total number of aliases in the *.ica* file. This line is generated by progeCAD.

```
nAliases=255
```

Alias=

The **Alias=** item is the one or more letter shortcut that represents the command. In the example above, **3A** is typed by the user on the keyboard, and progeCAD executes the **3DArray** command.

```
Alias=3A
```

LocalCommand= and GlobalCommand=

The **LocalCommand=** item represents the *localized* name of the command. IntelliCAD is available in a variety of languages, such as English, German, and Spanish. If you were using, say, the German release of IntelliCAD, then the German equivalent of the **3DArray** command would appear here.

```
LocalCommand=3DARRAY  
GlobalCommand=_3DARRAY
```

The **GlobalCommand=** item reports the internationalized version of the command; notice the underscore prefix. This is the (English) version of the command that works in all language releases of IntelliCAD. For example, **_3DArray** works in the German version.

Modifying Toolbars & Writing Macros

One of the easiest areas of progeCAD to customize, in my opinion, are the toolbars. Toolbars give you single-click access to almost any command and group of commands. Instead of hunting through progeCAD's menus (is the **Hatch** command under **Draw** or **Insert**?) or trying to recall the exact syntax of a typed command (was that **Viewpoint** or **VPoint**?), the toolbar lets you collect your most-used commands in convenient strips.

A group of commands executed by a single keystroke or mouse click is called a *macro*. Figuring in the time it takes to write and debug the macro, my rule-of-thumb — as you may recall from an earlier chapter — is to write a macro any time I repeat the same action more than thrice.

In This Chapter

- Customizing the look of toolbars.
- Creating new toolbars.
- Understanding the format of the .mnu file for toolbars, buttons, and flyouts.
- Writing toolbar macros.
- Sharing toolbars with others.

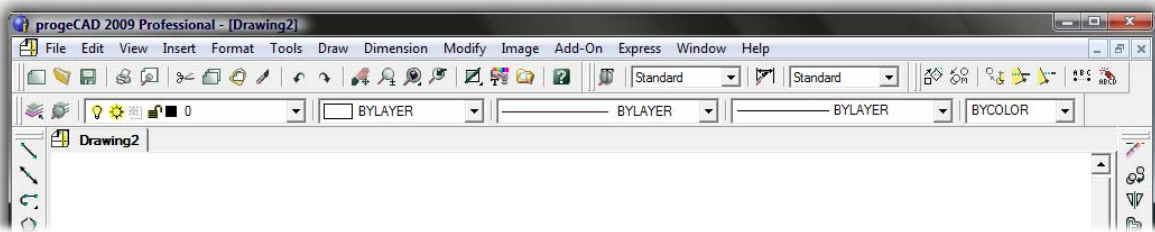
Customizing the Toolbar Look

There are two approaches to customizing toolbars. One approach rearranges the icons, perhaps creating new toolbars that hold oft-used commands or perhaps some of the commands not found on the toolbars that progeCAD displays by default.

The second approach writes macros that activate commands when toolbar buttons are clicked.

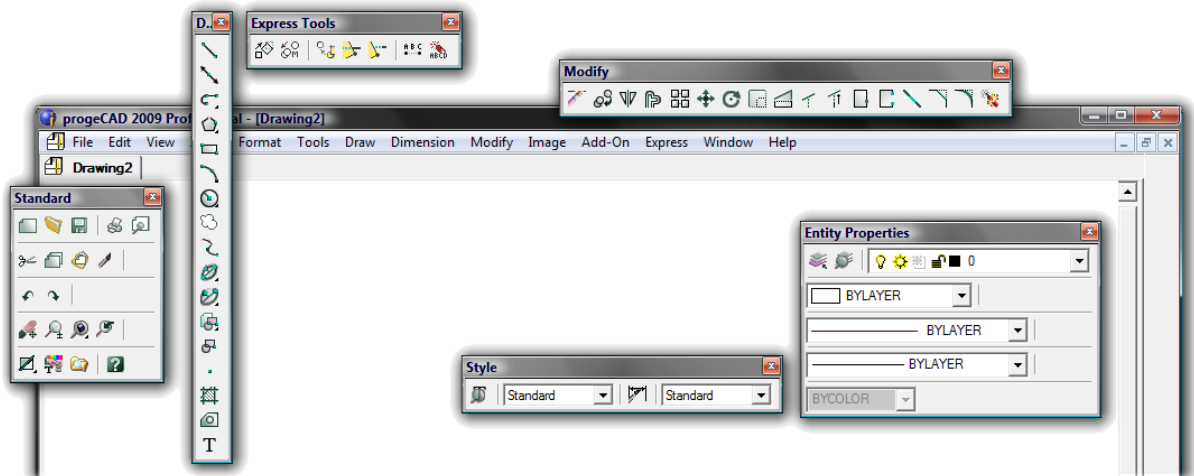
Rearranging Toolbars

When you start a fresh copy of progeCAD, it has several toolbars *docked* along the edges of the drawing area, as illustrated below. “Docked” means the toolbars are attached to the edge of the drawing area. When you move or change the size of the progeCAD window, docked toolbars move along.



Above: Toolbars docked along the edges of the drawing area.

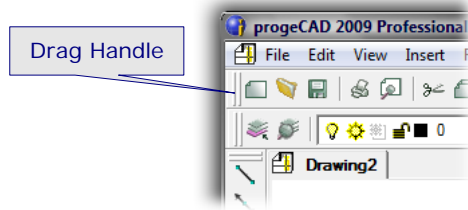
Below: Toolbars floating on the computer screen.



Toolbars don't have to be docked; they can also *float*. When toolbars float, they are independent of the progeCAD window. Move or resize the progeCAD window, and floating toolbars remain where they are; they can even be dragged onto a second monitor. Floating toolbars can be resized into square and long shapes, as illustrated above.

Dragging & Moving Toolbars

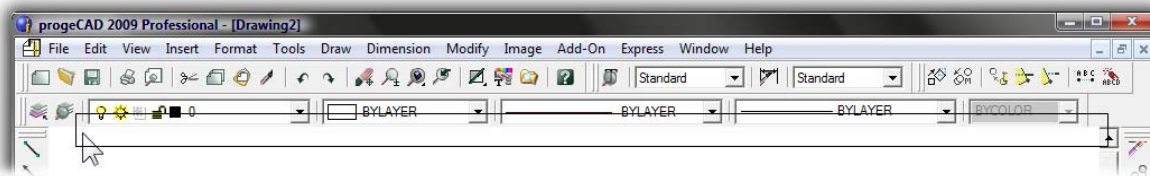
Look closely at one end of each toolbar, and notice the double-line (shown in the figure below). These are called *drag handles*. By dragging the toolbar by its handle, you move the toolbar around the screen.



You can relocate toolbars to other edges of the drawing area, or make the toolbars float. To move toolbars:

1. Drag the toolbar away from the edge of the drawing area.

Notice the thin, gray, rectangular outline. This is called the *dock indicator*, and is shown in the figure below. If you were to release the mouse button at this point, the toolbar would jump right back to its docked position.

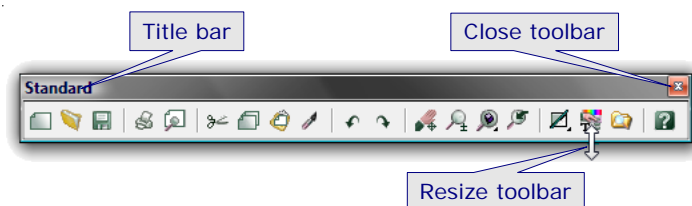


To prevent toolbars from docking, hold down the **Ctrl** key.

2. Drag the toolbar a bit further, and notice that the rectangular outline changes to a thicker line. This is called the *float indicator*.



3. Let go of the mouse button now, and the toolbar floats.
4. With the toolbar floating, you can move the toolbar by dragging it around by its title bar. To close a floating toolbar, click the red **x** button.

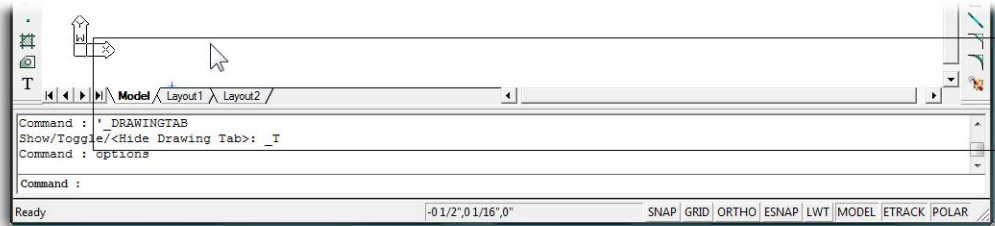


Additionally, you can resize the toolbar by grabbing at any of its edges. Notice the two-headed cursor in the figure above; it indicates that you can resize the toolbar, making it more rectangular or more square, as shown below.

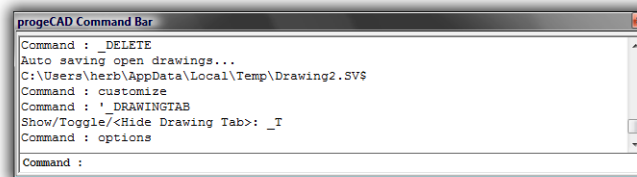


- To dock the toolbar again, drag it by its title bar back against one edge of the drawing area.

TIP Although not a toolbar, the Command Bar can be made to float and resize just like one. To float, drag an edge into the drawing area.



Once floating, you can move the Command Bar window by its title bar, and resize it by its edges — just like a toolbar.



To dock, drag the Command Bar window back into place.

The CmdBar command can force the command bar to float, dock, and hide:

Command: **cmdbar**
 Floating/Lower/Upper/<Show Command Bar>: (Enter an option.)

toggling the Display of Toolbars

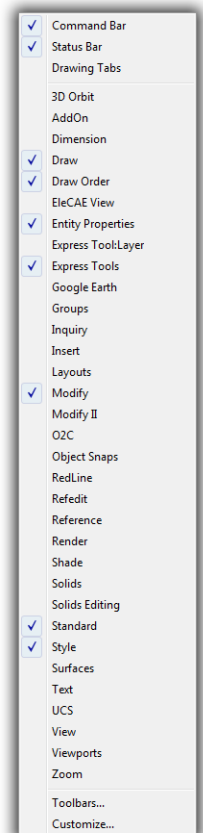
The display of toolbars can be turned off and on, as follows:

- Right-click any toolbar. Notice the shortcut menu that lists the names of all the toolbars. (Command Bar and Status Bar do not refer to toolbars.) The check mark, such as the one next to Standard, means the toolbar is displayed.
- To display a toolbar, select its name from the shortcut menu. Notice that the toolbar appears, and the shortcut menu disappears.

To turn on other toolbars, repeat steps 1 and 2.

- To turn off the display of a toolbar, repeat steps 1 and 2, but select toolbar names *with* check marks.

As an alternative, you can turn off floating toolbars by clicking the **x** in their upper right corner.



TIP To turn on (or off) *all* toolbars at once, use the **Toolbar** command, as follows:
: **toolbar**
Enter Toolbar name(s) to show or hide, ALL, or <?>: **all**
Show or Hide: **s**

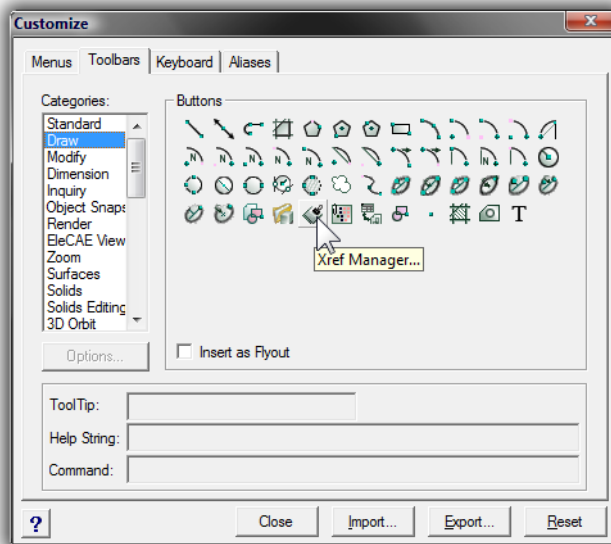
This command also turns on and off individual toolbars, which can be of use in macros and LISP routines — and even toolbar macros.

Creating New Toolbars

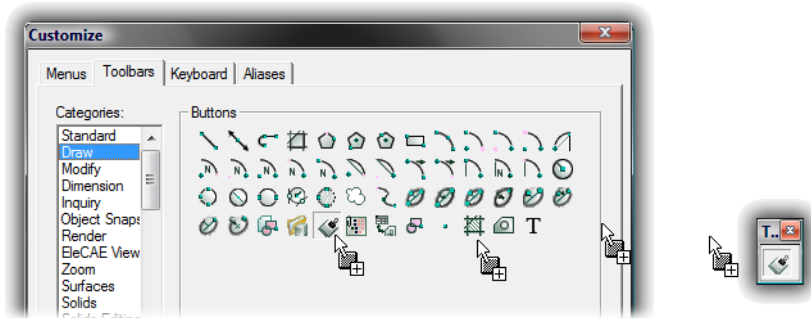
You can create new toolbars with the buttons (commands) of your liking. In this tutorial, we create a new toolbar that contains the XRef Manager button.

1. Right-click any toolbar. From the shortcut menu, select **Customize**. Notice the Customize dialog box. If necessary, choose the Toolbars tab.

(If no toolbar is visible, use the **Tools** menu: select **Customize** and then **Menu**.)



2. To create new toolbars, simply drag icons out from the Customize dialog box. For this tutorial, click on the Draw category, and then drag the XRef Manager button into the drawing.



Notice that progeCAD creates a new toolbar containing the button. And notice also that the cursor has a button and a plus (+) sign, to remind you that you are *adding* buttons.

TIP While the Customize dialog box is open, *all* toolbars are customizable — not just the one you created. This means you can add and remove buttons from the other toolbars, such as Draw, Standard, and so on.

4. You can drag buttons from the Customize dialog box into your new toolbar, or into any other existing toolbar. As you do, notice that the I-beam cursor helps you position the button among others on the toolbar.



5. To remove buttons, simply drag them out of the toolbar, and let go. Notice that the cursor has a button and an x sign, to remind you that you are removing buttons.



6. When done, click **Close** to dismiss the Customize dialog box. Your new toolbar acts like the any other toolbar in progeCAD.

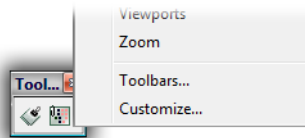
TIP You can remove buttons from a toolbar at any time, as follows: Hold down the **Shift** key, and then drag away the button from the toolbar.

To return removed buttons, go to **Tools | Customize | Toolbars**, and then repeat the process described earlier.

Renaming Toolbars

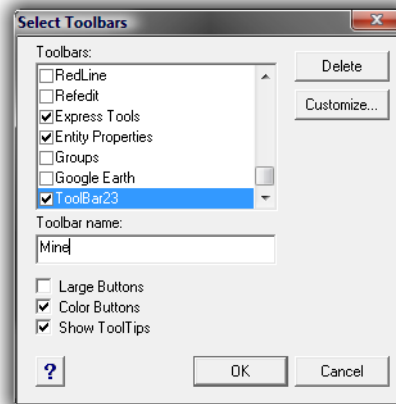
progeCAD gives your new toolbar a generic name, such as “Toolbar 5.” *Yawn!* You can, fortunately, change its name, as follows:

1. Right-click the toolbar, and select **Toolbars**.



2. Notice the Select Toolbars dialog box. Its primary purpose is to toggle the display of toolbars. But, we'll use it to change the name of your newly-created toolbar.

Scroll down the **Toolbars** list until you find the name of your new toolbar.

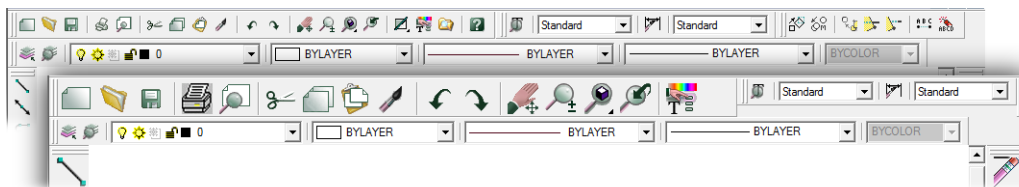


3. Select the toolbar name. In the **Toolbar Name** text box, change the name to something meaningful — like “My Favorite Tools.”
4. Click **OK**.

Changing Button Size, Color, and Tooltips

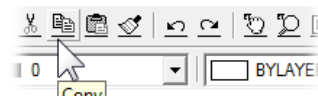
The buttons on toolbars are 16 pixels in size. In some cases, that may be too small if your eyesight is weak or if your monitor has a very high resolution, such as 1600x1200. You can make the buttons 50% larger, as follows:

1. From the menu, right-click any toolbar, and select **Toolbars**.
2. In the Select Toolbars dialog box, select **Large Buttons**.
3. Click **OK**. Notice that the buttons grow larger.



You may have noticed the other options in the Select Toolbars dialog box, which have — in my opinion — questionable usefulness:

- **Color Buttons** — when off, toolbar buttons are displayed in monochrome. This may help someone who is colorblind? Or maybe some people find the colors annoying? I don't know.



- **Show ToolTips** — when off, tooltips are no longer displayed when the cursor lingers over a toolbar button. I think this is a holdover from the days when computers were slow, and it took a lot of energy to show those little yellow tags. Who knows?

In either case, I always leave both options turned on.

TIP Make a big mistake in working with toolbars? Here is how to “fix” progeCAD by setting it back to its straight-out-the-box nature:

In the Customize dialog box, click **Reset** and all your changes disappear — for better or worse.

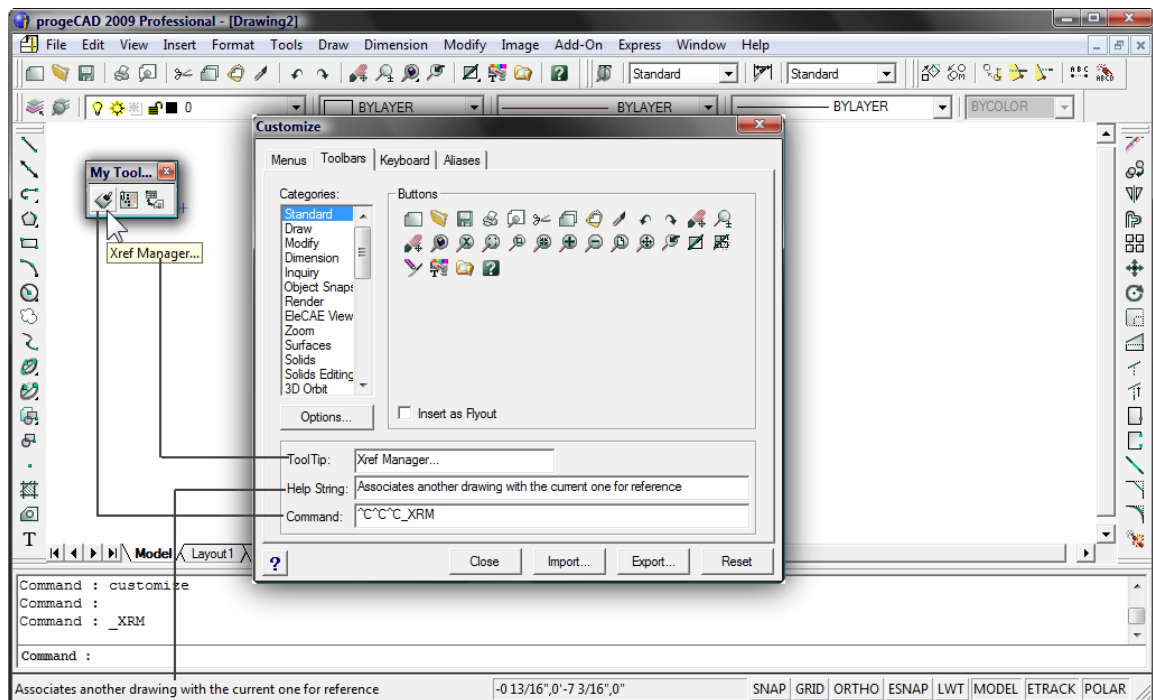
Writing Toolbar Macros

In addition to creating and changing toolbars, you can change the command(s) that lie behind each toolbar button, as well as the conditions under which the commands are executed.

When you click on a toolbar’s button, progeCAD executes the *macro* (a series of one or more commands) assigned to the button. Here is how to assign commands to buttons:

1. Bring back the Customize dialog box by right-clicking a toolbar, and then selecting **Customize**. (Or, enter the **Customize** command at the ‘:’ prompt.)
2. Select a button on your newly created toolbar. (Don’t select a button in the Customize dialog box — it won’t work). Notice that the bottom half of the Customize dialog box becomes active (ungrays itself).

The dialog box has several areas that correspond to IntelliCAD’s user interface, as shown by the brown lines in the figure below:



- **ToolTip** — text displayed by the button’s tooltip, a brief description of the button’s function.
- **Help String** — text displayed on the status bar, a longer description of the button’s function.

- **Command** — collection of commands (macro) executed by clicking the button. The figure below shows the macro that executes the **Align** command:

```
^C^C^C_ALIGN
```

Among the buttons:

- **Close** closes the dialog box, saving the changes you made.
- **Import** imports *.mnu* and *.mns* files generated by progeCAD and AutoCAD.
- **Export** exports a menu item in AutoCAD-compatible *.mnu* or *.mns* files.
- **Reset** changes the toolbars back to they way they first were.

Simple Macros

A simple macro consists an progeCAD command, prefixed by some unusual-looking characters:

```
^C^C^C_XRM
```

The characters have the following meaning:

- **^C** — *control character*. It imitates pressing **ESC** on the keyboard, canceling the command currently in progress. The carat (**^**) alerts progeCAD that this is a control character, and not a command.

(What does **C** have to do with the **ESC** key? In DOS-based software, we would press **CTRL+C** to cancel a command; the **C** was short for “cancel.” Confusingly, **CTRL+C** is the keyboard shortcut that means “copy to Clipboard”; in macros, it means “cancel”.)
- **^C^C^C** — cancels existing commands. Most macros start with three **^Cs** because some progeCAD commands are three levels deep, like **PEdit**. When the command is *transparent* (starts with the ' apostrophe), then you don't prefix the macro with the **Cancel** characters.
- **_** (underscore) — *internationalizes* the command. Prefixing the command name with the underscore ensures the English-language version of the command always works, whether used on a German, Japanese, or Spanish versions of progeCAD.
- **XRM** — command name. In macros, you type progeCAD commands and their options exactly the way you would type them on the keyboard at the ':' command prompt.

Nothing is needed at the end to terminate the command. progeCAD automatically does the “pressing **Enter**” for you.

Intermediate Macros

You can include commands, options, and prompts for user input in macros. For example, this macro draws an ellipse after the user specifies the center point and rotation angle:

```
^C^C^C_ELLIPSE;_C;\_R
```

Here is what the macro means:

- Recall that the three **^C** cancel any other command that might be active at the time.
- **ELLIPSE** is the name of the command, while the underscore (**_**) prefix internationalizes it.
- The semicolon (**;**) is just like pressing **ENTER** or the spacebar on the keyboard.
- The **C** is the Ellipse command's **Center** option. Just as you enter the abbreviation of options at the keyboard, so too you can use the same abbreviation in the macro — or you can spell out the entire option name, such as Center.
- The backslash (****) pauses the macro, waiting for the user to pick a point on the screen, or enter x,y-coordinates at the keyboard. Two backslashes in a row means that two picks are required.
- The **R** is the Ellipse command's **Radius** option.

Let's look again at the macro, this time in parallel with the command's prompts:

<code>^C^C^C</code>	<i>(Press Esc, Esc, Esc.)</i>
<code>_ELLIPSE;</code>	<i>: ellipse (Press ENTER.)</i>
<code>_C;</code>	<i>Arc/Center/<First end of ellipse axis>: c (Press ENTER.)</i>
<code>\</code>	<i>Center of ellipse: (Pick point.)</i>
<code>\</code>	<i>Endpoint of axis: (Pick point.)</i>
<code>_R</code>	<i>Rotation/<Other axis>: r (Press Enter.)</i>
	<i>Rotation around major axis: (Pick point.)</i>

A final semicolon (i.e. **ENTER**) and backslash (i.e. pause) are not needed at the end of the macro, because progeCAD automatically adds the equivalent of the symbology, and it no longer needs to wait for the user.

TIPS You can include aliases and LISP routines in menu macros. See the “Creating Keystroke Shortcuts & Aliases” and “Programming LISP” chapters to learn about them. You cannot, unfortunately, customize the icons displayed by toolbar buttons.

Toolbar Macros Are No Panacea

Toolbar macros are best suited for quick'n dirty programming. There are drawbacks, however, to using toolbar macros. The length of the macro is limited to a maximum of 255 characters. You are limited to using the icons supplied with progeCAD. The variety of control characters is extremely limited compared with IntelliCAD's other programming languages.

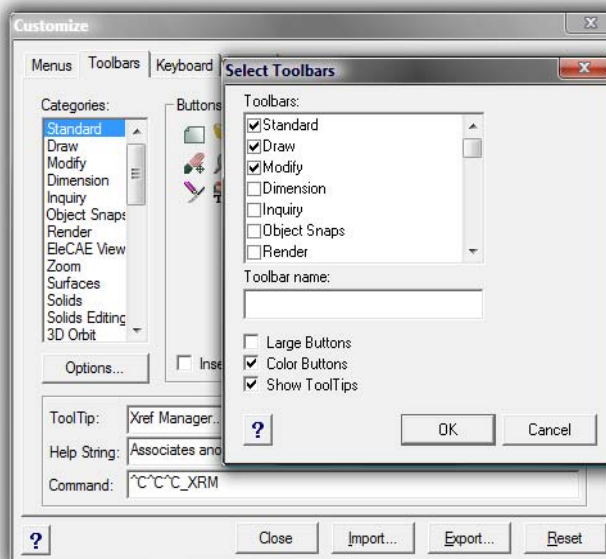
Despite these drawbacks, the toolbar is the fastest and most convenient way to minimize keystrokes and mouse clicks in progeCAD.

Sharing Toolbars

progeCAD allows you to share toolbars with other progeCAD and AutoCAD users. As the CAD manager, you've created an office standard for toolbars that need to be installed on all drafting computers running progeCAD. Here's how to save from your computer, and then load the toolbars onto the other computers.

Saving Toolbars

1. From the **Tools** menu, select **Customize**, and then choose the **Toolbars** tab.
2. Click **Export**.
3. Notice the Select Toolbars dialog box. progeCAD allows you to select which toolbars to export. Select one or more toolbar names, and then click **OK**.



4. Notice the Select Toolbar File dialog box. Enter a file name in the dialog box. If necessary, change the folder name.
5. Click **Save**. progeCAD saves the selected toolbars in an *.mnu* file.

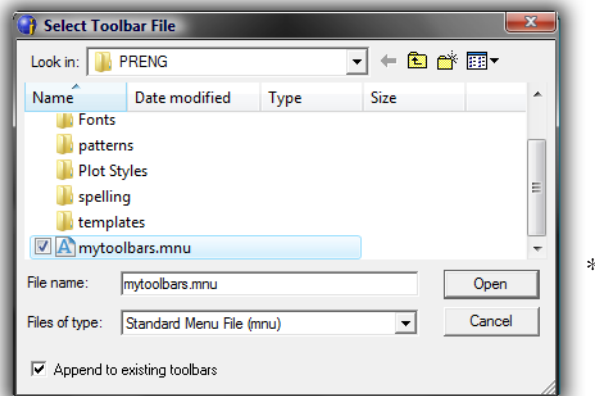
Importing Toolbars

With the toolbar(s) stored in an *.mnu* file, you can copy the file to other computers through the network, on USB key, or via an email attachment. The *.mnu* file can be imported into AutoCAD and progeCAD.

For progeCAD, follow these steps:

1. From the **Tools** menu, select **Customize**, and then select the **Toolbars** tab.
2. Click **Import**, and select the file name in the dialog box. (If necessary, change the folder name.)

3. Notice the **Append to existing toolbars** option, located at the bottom of the Select Toolbar File dialog box. You'll need to decide whether you want existing toolbars replaced by the new ones, or if the new ones should be added.



4. Click **Open**. Notice the newly added toolbars.

TIP If you used the **Append to existing toolbars** option, you may, unfortunately, notice double definitions. If this causes a problem:

Use the Customize dialog box's **Reset** button to clear the duplicates, and then repeat the **Import** command without the **Append** option turned off.

.mnu File Format

As shown in the previous section, the **Export** button of the Customize dialog box saves toolbars in AutoCAD's .mnu file format. It's curious switch: progeCAD exports its menus, aliases, and keystroke shortcuts in formats of its own design (see Chapters 2 and 4), but exports toolbars in AutoCAD's format.

After exporting toolbars from progeCAD, the first few lines of the .mnu file look like this:

```
***TOOLBARS
**STANDARD
TBAR_Standard [_Toolbar ("Standard", _Top, _Show, 111, 142, 1)]
ID_New [_Button ("New", , )]^C^C^C_NEW
ID_Open [_Button ("Open", , )]^C^C^C_OPEN
ID_Save [_Button ("Save", , )]^C^C^C_QSAVE

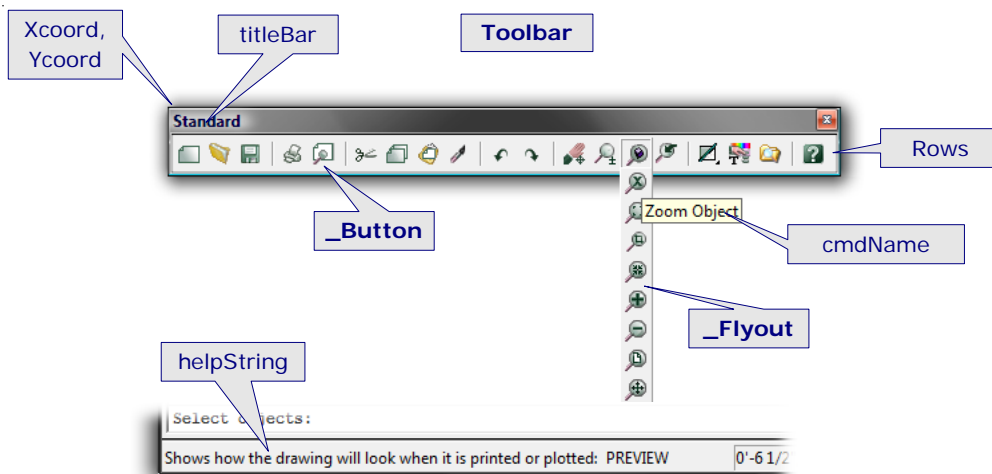
***HELPSTRINGS
TBAR_Standard [Standard]
ID_New [Creates a new drawing]
ID_Open [Opens an existing drawing]
ID_Save [Saves the current drawing]
```

General Format

The general format is:

```
***TOOLBARS
**name
TBAR_name [_Toolbar ("titleBar," defaultPosition, defaultVisibility, xCoord, yCoord, rows)]
    ID_cmdName [_Button ("cmdName", smallIcon, largeIcon)]macro
    ID_cmdName [_Flyout ("cmdName", smallIcon, largeIcon, _otherIcon, TBAR_name)]macro

***HELPSTRINGS
TBAR_name [name]
ID_cmdName [helpString]
```



In general, a toolbar definition consists of these items:

- Toolbar name and position information.
- A list of buttons and associated macros, and optional flyouts.

- Help string for each button.

For whatever reason, the help strings are listed separate from toolbar buttons; the two are linked by the `ID_cmdName`. If progeCAD can't find a match, no help is displayed on the status bar.

The tooltip wording is based on the `cmdName`.

Toolbar Format

```
***TOOLBARS
**name
TBAR_name [_Toolbar ("titleBar," defaultPosition, defaultVisibility, xCoord, yCoord, rows)]
```

***TOOLBARS

The section of the `.mnu` file that defines toolbars must start with `***TOOLBARS`.

**name

The `**name` item identifies the toolbar to other parts of the `.mnu` file. The `**`-prefix is an AutoCAD standard that identifies a subsection of the menu file, individual toolbars, in this case. (The `***`-prefix identifies major sections, such as toolbars, menus, and buttons.)

TBAR_name

The `TBAR_name` item identifies the toolbar for other parts of AutoCAD's `.mnu` file.

_Toolbar

The `_Toolbar` item indicates that the buttons following are part of a toolbar.

"titleBar"

`titleBar` specifies the words that appear on the toolbar's title bar. Recall that the title bar only appears when the toolbar is floating.

defaultPosition

The `defaultPosition` item specifies where the toolbar is positioned when progeCAD starts up. Valid values are:

- **Floating**.
- **Top** (docked at the top of the progeCAD window).
- **Bottom** (docked at the bottom of the window).
- **Left** (docked along the left edge).
- **Right** (docked along the right).

defaultVisibility

The *defaultVisibility* item specifies the visibility of the toolbar when progeCAD starts up. Valid values are:

- **Show** (toolbar is displayed).
- **Hide** (toolbar is not displayed).

xCoord and yCoord

The *xCoord* and *yCoord* item specify the position of the toolbar when progeCAD starts up.

The x-coordinate is of the left edge of the toolbar; the y-coordinate is of the top edge. The distance is measured in pixels from the left and top edges of the desktop.

rows

The *Rows* item specifies the number of rows the toolbar has when progeCAD starts up. When docked, rows = 1.

Button Format

```
ID_cmdName [_Button ("cmdName", smallIcon, largeIcon)]macro
```

ID_cmdName

The **ID_cmdName** item identifies the button; it associates the button with the help string, later in the *.mnu* file. Only the dash (-) and the underscore (_) can be used as punctuation.

_Button

The **_Button** item identifies this as a standard button, and not a flyout button.

cmdName

The *cmdName* item displays the tooltip when the cursor pauses over the button. Only the dash (-) and the underscore (_) can be used as punctuation.

smallIcon

The *smallIcon* item identifies the name of the small (16x15) bitmap icon. Because progeCAD does not allow you to change the icon associated with a button, this item is blank.

largeIcon

The *largeIcon* item identifies the name of the large (24x22) bitmap icon. Because progeCAD does not allow you to change the icon associated with a button, this item is blank.

macro

The *macro* item specifies the command(s) to execute when the user clicks the button. Toolbar macros use the same syntax and metacharacters as menu macros.

Flyout Button Format

```
ID_cmdName [_Flyout ("cmdName", smallIcon, largeIcon, _otherIcon, TBAR_name)]macro
```

The format of a flyout is identical to that of a button, with these differences:

`_Flyout`

The **_Flyout** item identifies this as a flyout button.

`_otherIcon`

The `_otherIcon` item determines which icon is displayed “on top” of the flyout. The possible values are:

- **OwnIcon** (the parent icon).
- **OtherIcon** (the most recently selected child icon).

`TBAR_name`

The **TBAR_name** item is the name of the toolbar to display as the flyout. It references any other toolbar name that starts with the `**`-prefix.

Help String Format

```
***HELPSTRINGS
TBAR_name [name]
ID_cmdName [helpString]
```

`***HELPSTRINGS`

The helpstring section must start with `***HELPSTRINGS`. In AutoCAD’s menu file, help strings are used by dropdown menu items, as well as by toolbar buttons.

`TBAR_name [name]`

The **TBAR_name [name]** item identifies the toolbar by name.

`ID_cmdName`

The **ID_cmdName** item identifies the name of the button.

`[helpString]`

The `[helpString]` item is the text of the help provided on the status bar.

Customizing Menus

In releases of AutoCAD prior to 2006, upon which progeCAD is based, the *.mnu* file controls the menu bar, all toolbars, “accelerator” keys (keyboard shortcuts), image tiles (an old form of dialog box), tablet overlays (used with digitizing tablets), and the rarely seen screen menu. (Since AutoCAD 2006, a different file is used, named *.cui*.) progeCAD is able to read AutoCAD’s *.mnu* files.

progeCAD is different. The default menus, toolbars, keyboard shortcuts, and aliases are hard-coded in the program. You makes changes to them via the Customize dialog box, as you saw for toolbars in the previous chapter.

In this chapter, we look at customizing the menus.

In This Chapter

- Modifying the menu bar.
- Editing macros.
- Editing help strings.
- Adding new menu items.
- Sharing menus.
- Importing AutoCAD menus.
- Understanding the menu file format.

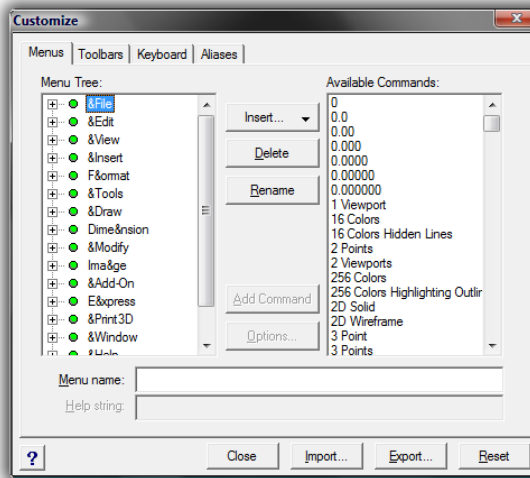
Modifying the Menu Bar

progeCAD lists many of its commands on the menu bar. Sometimes, however, you may want to change the menu. Adding commands to menus is particularly common for third-party developers. For example, Print3D adds its own menu to progeCAD.

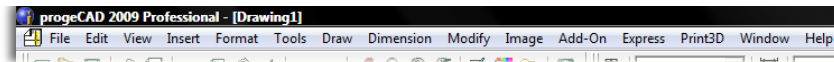
The following tutorial shows you how to add commands to the menu.

1. Menu customization takes place in the Customize dialog box. To open it, from the **Tools** menu, select **Customize**. (Alternatively, enter **Customize** at the 'Command:' prompt, or right-click any toolbar and then select **Customize**.)

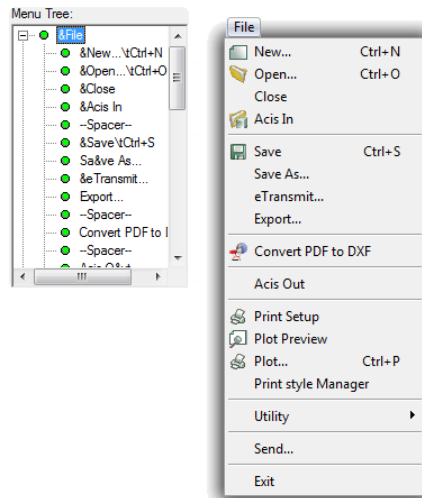
If necessary, click the **Menus** tab.



2. On the left, the Menu Tree shows the current menu structure. Notice that names like **File**, **Edit**, and so on match the names on progeCAD's menu bar.



3. Notice that each menu item in the Menu Tree has a + next to it. Click + next to **File** to reveal the items in the dropdown menu.



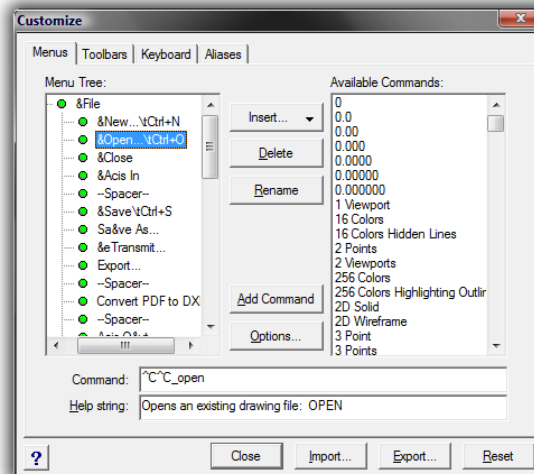
Opened up, the menu tree reveals a color code:

- Bright green dots mean the items appear in menus.
- Dark green dots mean the items appear in ActiveX containers.
- Bright red dots mean the items do not appear on the menu bar, but may appear in other menus.
- Dark red dots mean them items appear in context menus exclusively (shortcut menus).

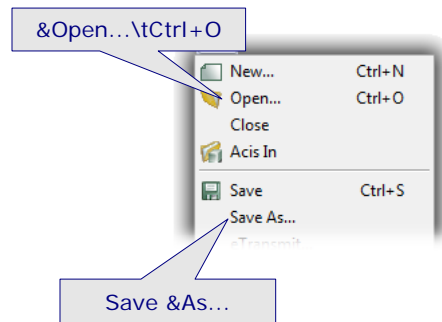
When you make changes, however, the colors of the dots do not, unfortunately, change — until *after* you close and then reopen the Customize dialog box.

Examining Menu Names

Let's look at how names are presented in menus. In the Customize dialog box, click on the File menu's Open item.



Notice that the “name” consists of several unusual characters, such as **&Open... \tCtrl+O**. The ones in boldface are called *metacharacters*. Let's look at how they relate to the menu:



Underline - &

The ampersand (&) is in front of the letter that should be underlined.

Underlined letters allow you to access commands from the keyboard using the ALT key. For example, to access the **New** command on the menu, you hold down the ALT key, press **F** (for **File**) and then press **N** (for **New**).

The convention is that the first letter of the name is underlined for mnemonic purposes. For example, **New**, **Open**, and **Save**. When there are two commands in one menu starting with the same letter, then you underline a different letter for the command appearing second. For example, the **File** menu has **Save As**, which appears as Name=**Save &As**...

Dialog Box - ...

The ellipsis (...) is for commands open dialog boxes. For example, **Save As...** displays a dialog box, whereas **Save** does not.

By itself, the ellipsis does nothing; it is merely a user interface element. You have to include code in the macro to open the dialog box.

Tab Separator - \t

The tab (\t) separates the menu item name from the keyboard shortcut, such as **Ctrl+N**. It serves to right-justify the shortcut from the command name.

New... and Ctrl+N

The words **New...** and **CTRL+N** are displayed by the menu. They are not, however, the command(s) that are executed when you select **New** from the **File** menu. To change the command, you need to edit the *macro*.

Caution: It is possible to change the keyboard shortcut with the Tools | Customization | Keyboard dialog box, so that the **CTRL**-key displayed by the menu does not match reality. For example, after deleting or changing the definition of **CTRL+N**, it still appears in the menu, but no longer opens a new drawing.

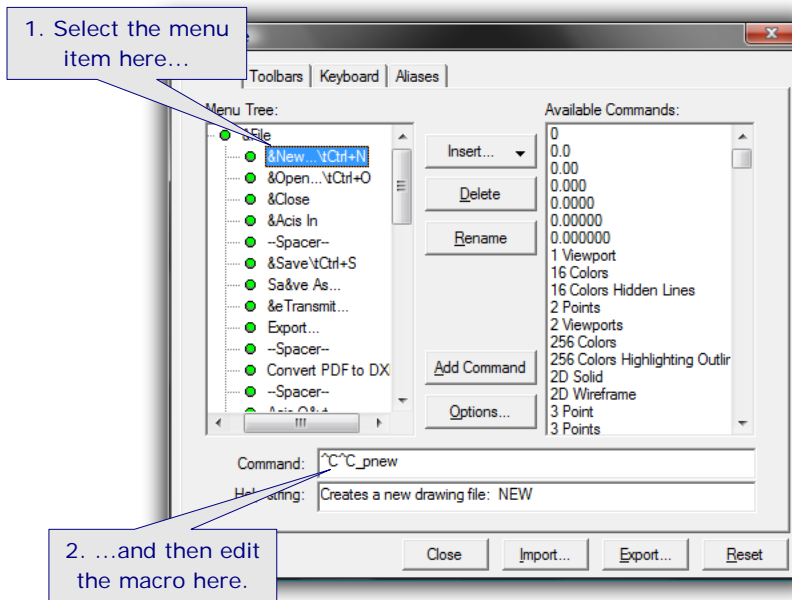
Editing Macros

To edit macros associated with menu items, follow these steps:

1. Select the menu item related to the macro.
2. Edit the macro, found in the **Command** text box at the bottom of the dialog box.

When user selects the item from the menu, this macro (series of commands) is executed. For example, selecting **New** from the **File** menu executes the **PNew** command (the New Drawing wizard).

Like the name, the macro contains metacharacters, as in `^C^C_pnew`. The macro syntax has the following meaning:



Cancel - ^C

The `^C` metacharacter means “cancel.” The caret (`^`) is the equivalent of the **Ctrl** key; together with `C`, `^C` is the same as pressing the **ESC** key to cancel a command.

The convention is to start every macro with three `^C` to cancel deeply nested commands, such as **PLine**.

Transparent - '

You do not, of course, prefix macros with `^C` if the command is to be operated *transparently*, such as `'_REDRAW`. The apostrophe metacharacter (`'`) means the command can be used during another command. Not all progeCAD commands operate that way.

Internationalize - _

The underscore (`_`) metacharacter “internationalizes” the command. progeCAD is available in a variety of (human) languages. By prefixing commands with the underscore, the command word is understood, even if it is used by the Spanish or German releases of progeCAD.

Enter - ;

The semicolon (`;`) metacharacter is equivalent to press the **ENTER** key. For example, the macro for the **View | Zoom | Zoom In** menu item looks like this:

```
'_ZOOM;2x
```

In the example above, the **Zoom** command accesses its **2x** option to zoom into the drawing. You typically use the semicolon to separate a command from its option.

The convention is to *not* include the semicolon at the end of the macro, because progeCAD automatically adds the **ENTER**.

Pause - \

The backslash (\) metacharacter pauses the macro for user input. In the example below, the macro pauses twice, because there are two backslashes in a row:

```
^C^C^C_DIMLINEAR;\_ROTATED
```

The **DimLinear** command waits for the user to pick two points, as follows (commands and options are in **blue**, pausing for user input are shown in **cyan**):

```
: _DIMLINEAR
ENTER to select entity/<Origin of first extension line>: (User picks first point.)
Origin of second extension line: (User picks first point.)
Angle/Text/Orientation of dimension line: Horizontal/Vertical/Rotated: _ROTATED
Angle of dimension line <O>:
```

The backslash metacharacter forces the macro to wait for either of two events to occur, and then carries on:

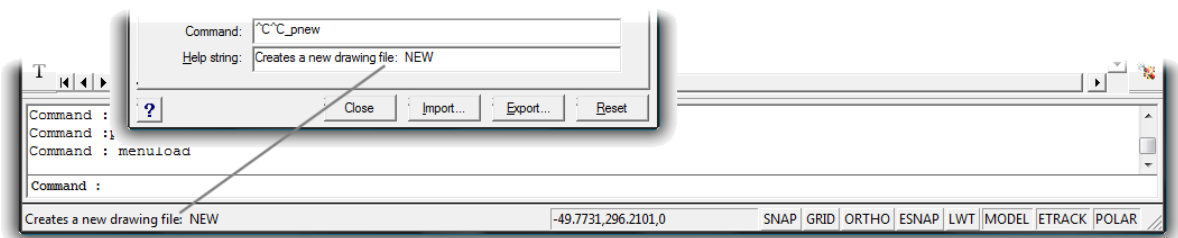
- The user picks a point on the screen.
- The user enters a value at the keyboard, and then presses **ENTER**.

TIP You can type commands, options, and metacharacters directly into the **Command** text box.

As an alternative, you can select commands from the **Available Commands** list, and then click **Add Command**. The advantage is that progeCAD automatically adds the ^C and _ metacharacters for you.

Editing the Help String

If you change the purpose of a menu item, then you may need to change the help string.



As shown by the illustration above, the help string is displayed on the status bar when the user selects a menu item. Edit the text in the **Help String** textbox.

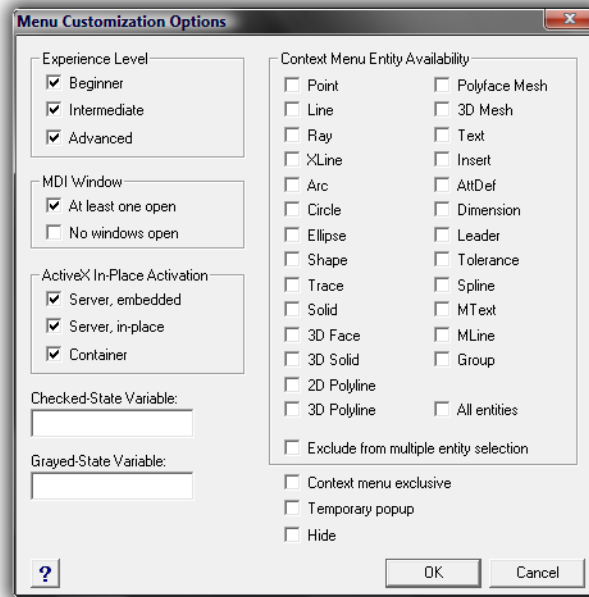
Changing Options

progeCAD lets you determine when and where a menu item appears. The options are shown when you select a menu item, and then click **Options**. The choices may, unfortunately, appear overwhelming to you. For the following figure, I selected **Dimensions** under **Menu Tree**, and then clicked **Quick Dimensions**.

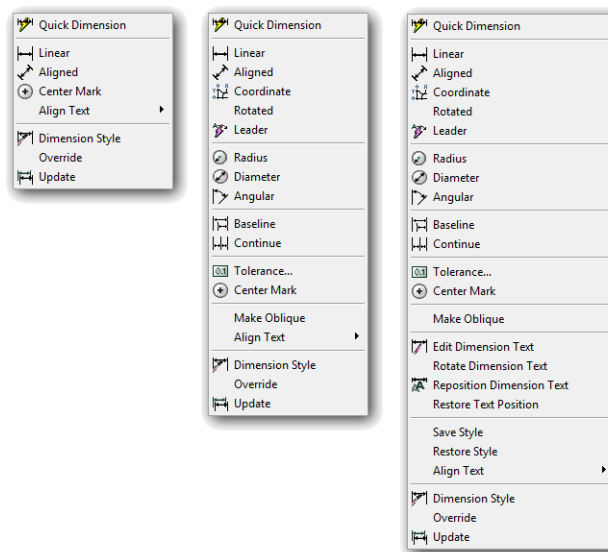
Let's take a look at what all these options mean.

Experience Level

- Beginner
- Intermediate
- Advanced



progeCAD has the ability to hide advanced commands from neophyte users. Use the **Config** command to set the experience level. When the Beginner option is not checked, for example, the associated menu item does not appear when **Experience level** is set to Intermediate or Advanced. The figure below shows the **Dimension** menu for the three experience levels:



Beginner

Intermediate

Advanced

MDI Window

- At least one open
- No windows open

The **MDI Window** settings determine when the menu item appears, depending on whether a window is open.

MDI is short for “multiple document interface,” a fancy term that means progeCAD can open more than one drawing at a time. (By the way, when a program can only open one document at a time, it has SDI, or “single document interface.”)

Most commands make no sense when no windows are open (when no drawings are loaded), so they disappear, as shown at the left, below:



Menu bar is sparse when no windows are open.

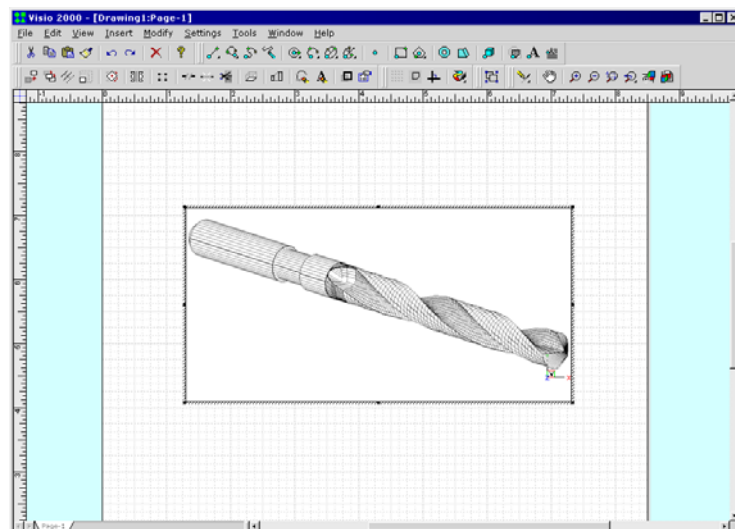


Full menu when at least one window is open.

ActiveX In-Place Activation

- Server, embedded
- Server, in-place
- Container

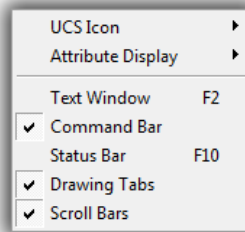
The ActiveX in-place activation settings determine which menu items are available when an progeCAD drawing is placed in another document. The figure below shows an progeCAD drawing in Visio, with some of the progeCAD toolbars and the menu bar replacing those of Visio.



Checked-State and Grayed-States Variables

- Checked-State Variable
- Grayed-State Variable

The **Checked-State Variable** option places a check mark in front of a menu item, as shown by the items on the **View | Display** menu, below. The convention is that the check mark indicates a value is turned on.



Whether the check mark is displayed depends on the value of the related system variable. When it is on (1), then the check mark is shown; when off (0), the check mark is not shown.

You can use any system variable in progeCAD, although the check mark is mostly used with *toggle* system variables. (Toggle system variables are ones that are either on or off; they don't have any other value.)

The **Grayed-State Variable** turns the menu item gray, depending on the value of the system variable. The convention is that a gray menu item indicates it is not available. Grayed-State can make use any system variable in progeCAD.

Both Checked-State and Grayed-State make use of the following metacharacters:

Value - &

The ampersand (&) lets you access specific values of a system variable. For example, `UNDOCTL&0` means that the **Undo** menu item is gray when system variable **UndoCtl** equals 0. To find the valid values, look up the online help for system variables.

Not - !

The exclamation mark (!) means “not.” It is useful for system variables that are not toggles; when a system variable has more than two settings, you can use ! when you want the menu item to be gray only for one setting.

For example, `!VIEWMODE&1` means that the **Pan** menu item would be gray when ViewMode is *not* 1 (1 = perspective mode is on). In other words, you can't pan in a perspective view.

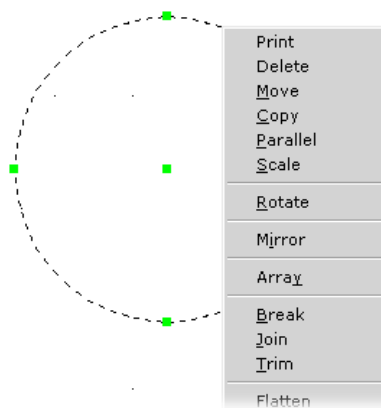
Context Menu Entity Availability

- *Individual entities*
- All Entities
- Exclude from multiple entity selection

Context Menu Entity Availability is used in conjunction with the **Modify** menu. When the user selects one or more objects, and then presses the right mouse button, progeCAD displays a context menu that is essentially the **Modify** menu; see figure below.

If the command works with all entities, then select the **All Entities** option.

When the command works with specific entities only, then select those entities. For example, the **Fillet** command works with lines, arcs, circles, and other entities, but not with traces, dimensions, and text.



The context menu appears when you right-click a selected object.

To make the command available only when a single entity is selected, use the **Exclude from multiple entity selection** option.

TIP Only commands appearing in the **Modify** menu appear in the context menu.

Miscellaneous

- Context Menu Exclusive
- Temporary Popup
- Hide

The **Context Menu Exclusive** option means the command appears only in the context menu, as described above under Context Menu Entity Availability.

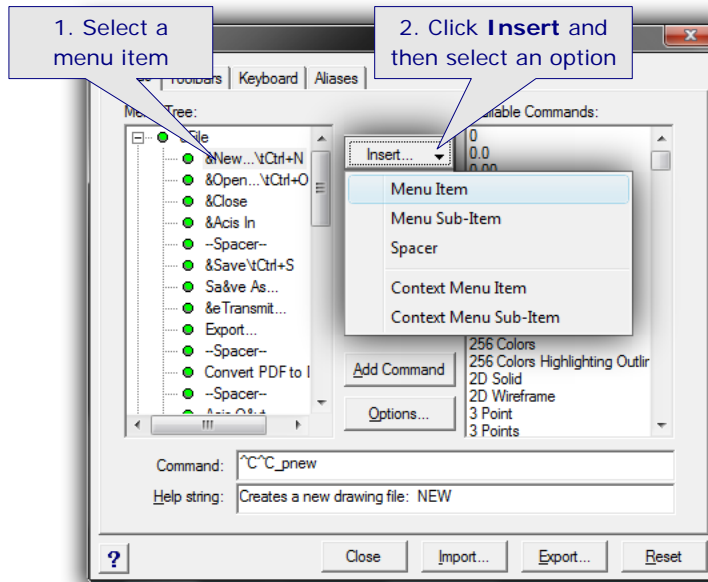
I have no idea what the **Temporary Popup** option is for.

The **Hide** option hides menu items from view. This option is useful for when you are working on a macro that you do not want users to use, yet.

Adding New Menu Items

To add a new menu item:

1. Under **Menu Tree**, select the menu item *above* which the new one is to appear.



2. Click **Insert**, and select one of the options:

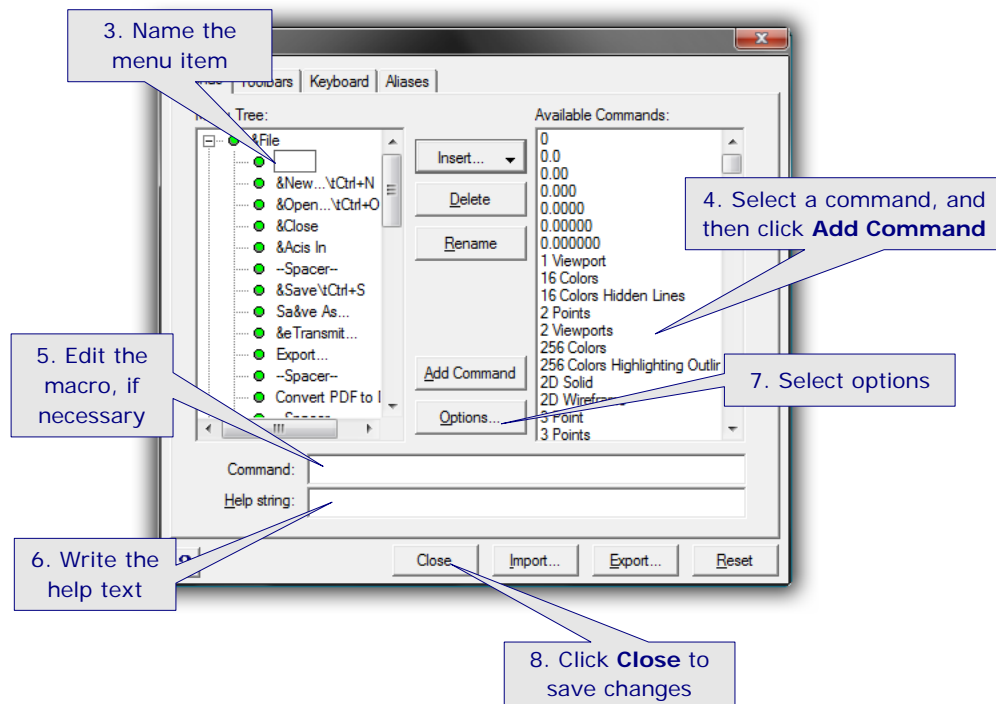
- **Menu Item** inserts a menu item.
- **Menu Sub-Item** creates a parent menu, from which you add submenus.

Careful: Existing menu items are turned into parent menus, and lose their associated macro.

- **Spacer** places a gray line in the menu, to separate groups of menu items.

The **Context Menu Item** and **Context Menu Sub Item** options puzzle me, because they don't seem to work.

3. Notice that a new menu item is inserted in the Menu Tree, but it has no name. Name it.
4. Specify the macro. The easy way to do this is to select a command (from **Available Commands**) and then click **Insert Command**.
5. Write the help text. This is the text that appears on the status line.



7. If there are special conditions as when and how the menu item is to appear, click **Options**. The default settings are that new menu items appear when the Advanced experience level is selected, and when at least one window is open.
8. To save your valuable work, click **Close**.

Deleting Menu Items

To remove a menu item, select it in the Menu Tree, and then click **Delete**. progeCAD asks if you really want to do this: “Are you sure you want to remove the item?” You can delete individual menu items, as well as submenus.

Did you make a horrible mistake? There is no undo button. Instead, click the **Reset** button to return the menus to their fresh-out-of-the-box nature. You will, however, lose any editing changes you may have wanted to keep.

The workaround is to use the **Export** button to export the menu as an *.icm* file, and then perform major surgery in the Customization dialog box. If you make a big mistake, click the **Import** button *and* make sure **Append to Current Menu** option is turned *off*.

ICM Menu File Format

When you use the Customize dialog box's **Export** command, progeCAD saves the menu as an *.icm* file (no, not short for “Inter Continental Missile,” but “Intelli Cad Menu”). The first few lines of a typical *.icm* file are shown below:

```
[IntelliCAD Custom Menu File]
nMenuItems=283
[Mnultem-0]
Name=&File
TearOffName=POPI
Command=POPI
Visibility=12
[Mnultem-1]
Name=&New...    Ctrl+N
Command=^C^C^C_NEWWIZ
HelpString=Creates a new drawing
Visibility=191
SubLevel=1
```

The *.icm* file is not documented, so the following sections explain each entry.

nMenuItems

The **nMenuItems** item is a count of the total menu items in the file. It is generated by progeCAD. Changing the number from a valid value causes progeCAD to crash when importing the modified *.icm* file.

Name

The **Name** item is the name displayed on the menu bars and menu items:

- On the menu bar, Name=**&File**
- On menu items, Name=**&New...** \t **Ctrl+N**



(The \t does not appear. It indicates the presence of a tab space.) The syntax has the following meaning:

Alt-Shortcut - &

The ampersand (&) is in front of the letter that should be underlined. Underlined letters allow you to access commands from the keyboard using the ALT key. The convention is that the first letter of the name is underlined for mnemonic purposes. When there are two commands in one menu starting with the same letter, then you underline a different letter for the command appearing second. For example, the **File** menu has **S**ave and **S**ave **A**s, which appears as **Name=Save &As...**

Dialog Box - . . .

The ellipsis (. . .) is used when the command opens a dialog box. In itself, the ellipsis does nothing. For example, **Save As...** displays a dialog box, whereas **Save** does not.

Right-Justified - \t

The tab (not visible) separates the menu item name from the keyboard shortcut, such as **CTRL+N**.

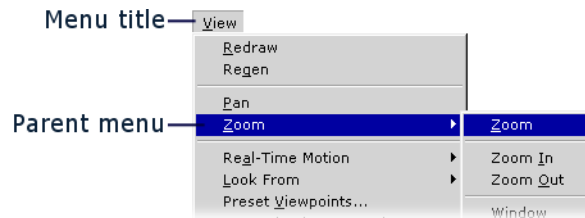
Caution: It is possible to change the keyboard shortcut with the Tools | Customization | Keyboard dialog box, so that the **CTRL**-key displayed by the menu does not match reality. For example, after deleting or changing the definition of **CTRL+N**, it still appears in the menu, but no longer opens a new drawing.

TearOffName

The **TearOffName** item is, I believe, a placeholder that tells progeCAD this menu item is not a command, but another menu element, such as a menu title or a parent menu.

`TearOffName=POP1`

The figure below shows both, neither of which executes a command; rather both display “another menu.”



To create a parent-child menu, see **SubLevel**, later in this section.

AutoCAD uses the `POPn` notation, which is short for “popdown menu,” to identify menus, allowing one menu to be called by another. The equivalent menu item in progeCAD, `Command=POP1`, does not, however, work.

Command

The **Command** item is the name used by the menu bars and menu items:

- On the menu bar, **Command=POP1**
- On menu items, **Command=^C^C^C_NEWWIZ**

For the menu bar, I don’t know what the “`Command=Pop1`” macro means. I suppose that it executes itself, or may be dummy expression.

For the menu items, **Command** is the *macro* executed by selecting the menu item. (A “macro” is one or more commands executed in sequence.) The macro syntax has the following meaning:

Cancel - ^C

The **^C** metacharacter means “cancel.” The caret (^) is the equivalent of the **CTRL** key; together with **C**, **^C** is the same as pressing the **ESC** key to cancel a command. (**CTRL+C** harkens back to the days of DOS, when it meant “cancel.” Note that this works only in macros; at ‘:’ command prompt, **CTRL+C** means “copy to the Clipboard.”)

```
Command=^C^C^C_PNEW
```

The convention is to start every macro with three **^C** to cancel deeply nested commands, such as **PLine**. (You do not, of course, prefix macros with **^C** if the command is to be operated transparently, such as `Command='_REDRAW`.)

Internationalize - _

The underscore (_) metacharacter “internationalizes” the command.

```
Command=^C^C^C_PNEW
```

progeCAD is available in a variety of (human) languages. By prefixing commands with the underscore, the command word is understood, even if it is used by the Spanish or German releases of progeCAD.

Enter - ;

The semicolon (;) metacharacter is equivalent to press the **ENTER** key. For example, the macro for the **View | Zoom | Zoom In** menu item looks like this:

```
Command='_ZOOM;2x
```

In the example above, the **Zoom** command accesses its **2x** option to zoom into the drawing. You typically use the semicolon to separate a command from its option.

The convention is to *not* include the semicolon at the end of the macro, because progeCAD automatically adds the **ENTER**.

Pause - \

The backslash (\) metacharacter pause the macro for user input. In the example below, the macro pauses twice, because there are two backslashes in a row:

```
Command=^C^C^C_DIMLINEAR;\_ROTATED
```

The **DimLinear** command waits for the user to pick two points. The backslash metacharacter forces the macro to wait for either of two events to occur, and then carries on:

- The user picks a point on the screen.
- The user enters a value at the keyboard, and presses **ENTER**.

Visibility

The **Visibility** item is a bit code that specifies when a menu item is displayed.

```
Visibility=| | |
```

The bit codes are:

Bit Code	Comment
Experience Level	
1	Beginner
2	Intermediate
4	Advanced
MDI Window	
8	At least one open
16	No windows open
ActiveX In-Place Activation	
32	Server, embedded
64	Server, in-place
128	Container
Other	
256	Context menu exclusive
512	Hide
1024	Temporary popup

Bit codes are added together to indicate multiple settings. For example, if a menu item were to be available for all three experience levels, the value of the bit code would be 7 (1 + 2 + 4).

Experience Level

The visibility settings for experience level indicate whether the menu item appears in menus set up for **Beginner**, **Intermediate**, and **Advanced** levels. (The experience level is set with the **Tools | Options | General** command.) For example, the **Ellipse** command appears for all three levels, but **Elliptical Arc** appears only when experience level is set to **Intermediate** and **Advanced**.

MDI Window

The MDI window settings determine when the menu item appears, depending on whether a window is open. MDI is short for “multiple document interface.” There are two bit codes:

- **8** means the menu item appears when at least one window is open.
- **16** means the menu item appears even when no window is open.

Most commands make no sense when no windows are open (when no drawings are loaded), so they disappear.

ActiveX In-Place Activation

The ActiveX in-place activation settings determine which menu items are available when an progeCAD drawing is placed in another document. There are three bit codes:

- **32** means the command appears when the drawing is an embedded server.
- **64** means the menu item appears when the drawing is an in-place server.
- **128** means the menu item appears when the drawing is a container.

Other

The other settings determine when the menu item appears in three miscellaneous cases. There are three bit codes:

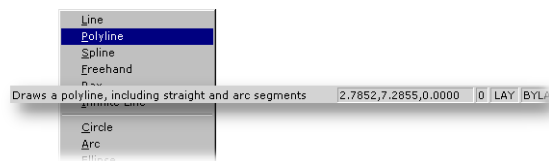
- **256** means the menu item appears only in context menu (a.k.a. shortcut menu).
- **512** means the menu item is hidden.
- **1024** means the menu item appears only in a temporary popup.

HelpString

The **HelpString** item displays a line of text on the status line explaining the menu item. For example,

```
Name=&Polyline
Command=^C^C^C_POLYLINE
HelpString=Draws a polyline, including straight and arc
```

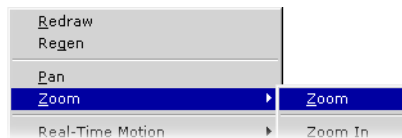
displays the text “Draws a polyline, including straight and arc” on the status line when **Polyline** is selected.



SubLevel

The **SubLevel** item determines if the item appears on the parent menu or the child menu; or, if it appears in the main menu or the submenu. There are two values in common use, but you can use additional digits for deeper menus:

- **1** means the item appears in the parent menu.
- **2** means the item appears in the child (or sub) menu.
- **3** means the item appears in the sub-sub menu.
- and so on.



The following code shows the **Zoom** submenu illustrated above:

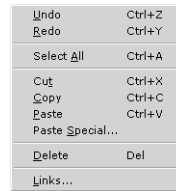
```
Name=&Zoom
TearOffName=POP9
SubLevel=1

Name=&Zoom
Command='_ZOOM
SubLevel=2
```

AddSpacerBefore

The **AddSpacerBefore** item draws a gray line to separate menu items, as shown below.

AddSpacerBefore=1



The value, **1**, seems to be a dummy. It appears to make no difference if the value is 0, 1, 2, or 3.

EntityVisibility

The **EntityVisibility** item is used in conjunction with the **Modify** menu. When the user selects one or more objects, and then presses the right mouse button, progeCAD displays a context menu that is a copy of the **Modify** menu. The commands listed on the menu depend on several conditions:

- The setting of the **EntityVisibility** bit code(s), as listed in the table below.
- Whether bit code -2147483648 is set (exclude from multiple entity selection).
- The setting of **Visibility** bit codes.

For example, the **Copy** command works with all entities, but the **Break** command does not.

Bit Code	Meaning
1	Point
2	Line
4	Ray
8	Xline
16	Arc
32	Circle
64	Ellipse
128	Shape
256	Trace
512	Solid
1 024	3D face
2 048	3D solid
4 096	2D polyline
8 192	3D polyline
16 384	Polyface mesh
32 768	3D mesh
65 536	Text
131 072	Insert (block)
262 144	Attribute definition
523 288	Dimension

1 048 576	Leader
2 087 152	Tolerance
4 194 304	Spline
8 388 608	Mtext
16 777 216	Mline
33 554 432	Group
536 870 911	All entities
-2147 483 648	Exclude from multiple entity selection

Note that “Exclude from multiple entity selection” carries a negative value.

ChekVar

The **ChekVar** item places a checkmark in front of a menu item. The convention is that the check mark indicates a value is turned on. In this case, the display of fills is turned on.

```
ChekVar=FillMode
```

Whether the check mark is displayed depends on the value of the system variable. When **FillMode** is on (1), then the check mark is shown; when **FillMode** is off (0), the check mark is not shown.

You can use any system variable in progeCAD, although the check mark is mostly used with *toggle* system variables. (Toggle system variables are ones that are either one or off; they don’t have any other value.) There are some cases where other system variables are used with **ChekVar**, such as showing which entity snap modes are turned on.

ChekVar uses the same metacharacters as **GrayVar**; see the next item. **ChekVar** is short for “CHEcK VARIable” (notice there is a **c** missing in “chek,” so don’t misspell it!).

GrayVar

The **GrayVar** item turns the menu item gray, depending on the value of the system variable.

```
GrayVar=!ViewMode&I
```

The convention is that a gray menu item indicates it is not available. **GrayVar** can make use any system variable in progeCAD, along with the following metacharacters:

Value - &

The ampersand (&) lets you access specific values of a system variable. For example:

```
Name=&Undo
GrayVar=UNDOCTL&0
```

means that the **Undo** menu item is gray when system variable **UndoCtl** equals 0. To find the valid values, look up the online help for system variables. **UndoCtl**, for example, uses these bit codes: 0=undo turned off; 1=undo turned on; 2=single undo; 4= auto undo.

Not - !

The exclamation mark (!) means “not.” It is useful for system variables that not toggles; when a system variable has more than two settings, you can use ! when you want the menu item to be gray only for one setting. For example:

```
Command='_PAN  
GrayVar=!VIEWMODE&I
```

means that the **Pan** menu item is gray when `ViewMode` is *not* 1 (1 = perspective mode is on). In other words, you can't pan in a perspective view.

Customizing Linetypes

Linetypes in progeCAD can be *simple* or *complex*.

- **Simple linetypes** are one-dimensional, consisting of lines, gaps, and dots put together in a variety of patterns.
- **Complex linetypes** include 2D shapes and text, such as railroad tracks and — CW — for a coldwater line.

DHWRX2	Domestic Hot Wtr Return	
DHWX2	Domestic Hot Water	
DIVIDE		
DIVIDE2		
DIVIDEX2		
DOL013	07b013 Dotted line	
DOT		
DOT16(1/2 of Dot8).....	

progeCAD comes with a number of linetypes, stored in the *icad.lin* and *icadiso.lin* files found in the *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG* folder.

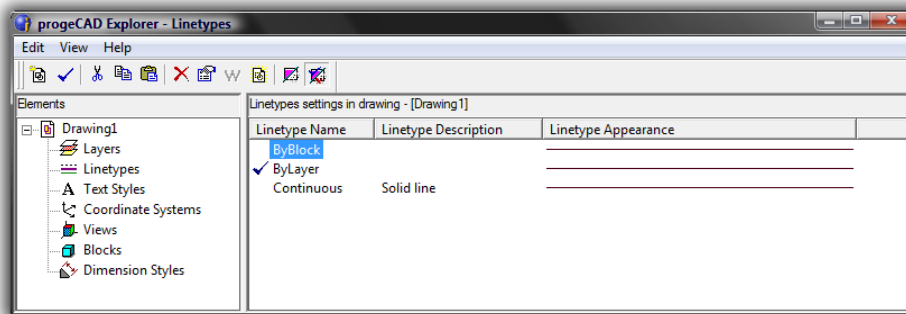
In This Chapter

- Commands and system variables affecting linetypes.
- Special case of polylines.
- Compatibility with AutoCAD.
- Customizing linetypes.
- Editing linetype definitions.
- Testing new linetype.
- Creating linetypes with text editors.
- Understanding the linetype format.
- Complex (2D) linetypes.

Commands That Affect Linetypes

Linetypes are defined in files external to progeCAD, with an *.lin* extension. It's always been a source of irritation to me that I gotta load the file into the drawing *before* I can use any linetype. The **Linetype** command loads the linetypes, and lists the linetypes already loaded.

The **ExpLTypes** command (short for “explore linetypes”) displays progeCAD Explorer with linetypes. In addition, it loads linetypes, creates new linetypes, renames and deletes them, and purges unused linetypes from the drawing.



Like colors, you can apply linetypes to individual objects with the Entity Properties toolbar. Or, through the Layer command, you can assign all objects located on a layer to have the same linetype.

Like text, linetypes are tricky to size. You have to size the gaps and dashes just the right way. Too small, and the line looks solid (but takes a suspiciously long time to redraw). Too large, and the line looks solid, too. It's the **LtScale** command (short for “linetype scale”) that lets you set the scale of the linetype. Typically, the scale used for text and dimensions and hatch patterns also applies to the linetype. Nice, eh?

System Variables that Affect Linetypes

Because linetypes are affected by scale, paper space becomes a problem. A linetype scale that looks fine in model space is going to look wrong in paper space. The solution comes with the **PsLtScale** system variable (short for “paper space linetype scale”). Its job is to assign the viewport's scale factor to linetypes.

There are a couple of other system variables that relate to linetypes. **CeLtype** (short for “current entity linetype”) holds the name of the linetype currently in effect. **LtScale** stores the current linetype scale factor (default = 1.0).

The Special Case of Polylines

Then there's the trick when it comes to polylines. To understand the problem, understand how progeCAD generates a linetype. The software attempts to apply linetypes as nicely as it can, based on the length of the object and the linetype scale factor. Essentially, it starts at one end of the object and works its way to the other end. Then progeCAD centers the linetype pattern so that it looks nice and even at both ends. You never get the linetype abruptly ending midway through at the end of a line.

Consider, then, the polyline. While it looks like one long connected line-arc-spline, it has many vertices, even when you do not see them. progeCAD faithfully restarts the linetype pattern each time it encounters a vertex. When the vertices are close together, progeCAD never gets around to restricting the pattern, resulting in a solid or continuous line. This would drive some people nuts, like cartographers who use polylines for drawing contours.

The solution is the **PlineGen** system variable (short for “polyline generation”). When set to off (the default), progeCAD works as before, generating the linetype from vertex to vertex. When changed to on (1), progeCAD generates the linetype from one end of the polyline to the other end — ah, instant relief!

Compatibility with AutoCAD

progeCAD includes all of AutoCAD’s simple linetypes, and a few of AutoCAD’s complex linetypes. If shapes in complex linetypes are displayed as ? (question marks), this means progeCAD was unable to find the *ltypeshp.shx* file. The result is shown below:

Fenceline 1 _____? _____? _____

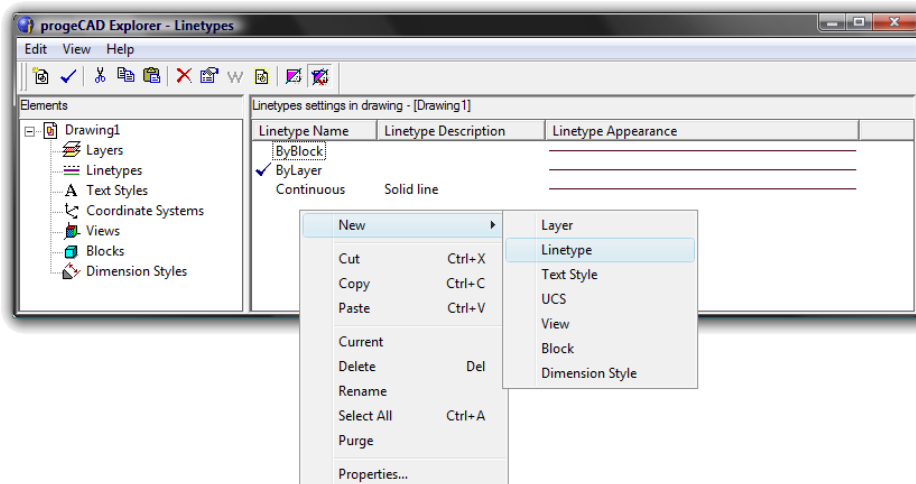
Customizing Linetypes

progeCAD has three ways of creating a custom linetype: (1) through the progeCAD Explorer; (2) at the command prompt; and (3) with a text editor. Let’s look at the first one first.

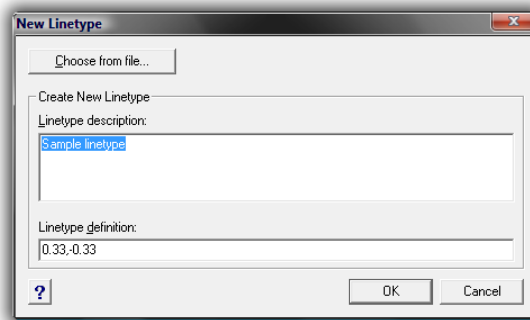
progeCAD Explorer

Here is how to create a custom linetype with Explorer:

1. From the **Format** menu, select **Explore Linetypes**.
(Or, select **Tools | progeCAD Explorer** , and then select **Linetypes**. Or, enter the **ExpLTypes** command.)
2. Right-click anywhere in the right-hand pane. From the shortcut menu, select **New | Linetype**.



Notice that the New Linetype dialog box. It has two fields: (1) **Linetype description**; and (2) **Linetype definition**. Naming the linetype comes later.



3. Enter a linetype *description*, such as “Sample linetype.”

The description is any phrase that will be useful in the future for identifying the linetype. Other examples include:

- Boundary
- Domestic Hot Wtr Return
- Steam Condensate
- Long dashed short dashed line 2mm

4. Enter the linetype *definition*, such as 0.33,-0.33.

The definition consists of numbers and punctuation only. progeCAD prevents you from entering text, which means you cannot use the Explorer to create complex lintypes. Follow this code:

- **Positive numbers indicates dashes** — for example, 0.25 means a dash 0.25 units long.
- **Negative numbers indicates gaps** — for example, -0.1 is a gap 0.1 units long.
- **Zeros draw a dot** — a 0 is a single dot.
- **Commas** — separate the codes. For example, .25,-.1,0,-.1

And follow this rule:

- You can't use the same code twice in a row. It just doesn't make sense to have two gaps or two lines in a row, does it? Instead, code that gap or line twice as long.

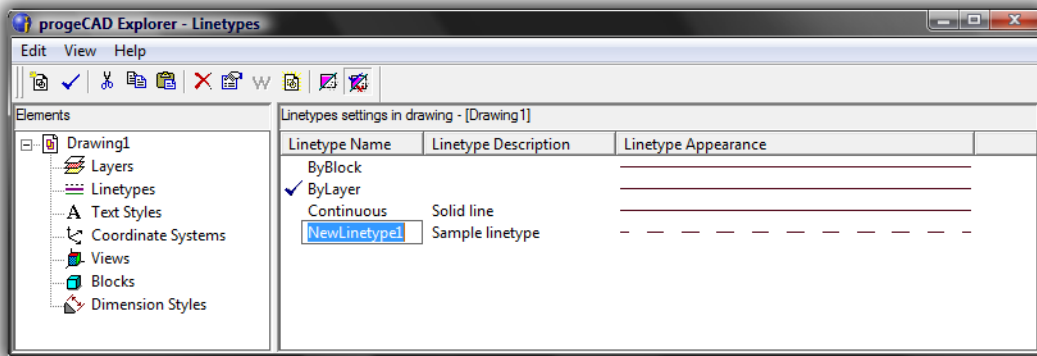
For this tutorial, I entered:

```
0.33,-0.33
```

This draws a line 0.33 units long, followed by a gap 0.33 units long (specified by the negative sign). The pattern will repeat itself automatically.

5. Click **OK**.

Notice that the description you gave the linetype appears under **Linetype Description**, while the code you entered appears as dashes and gaps under **Linetype Appearance**.



6. progeCAD gives a generic name to the linetype, **NewLinetype1**. You can edit it to a more meaningful name.

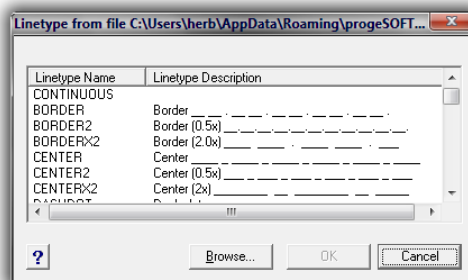
Later in this chapter, you learn how to create linetypes with the **Linetype** command, and how to edit the *icad.lin* file directly.

Editing the Linetype Definition

You may have noticed that I did not mention the two buttons in New Linetype dialog box. I'll get to them now.

1. Click the **New Item** button on the toolbar. (Alternatively, from Explorer's menu bar, select **Edit | New | Linetype**.)
2. In the New Linetype dialog box, click **Choose from file**.

Notice Linetype Front dialog box. It lists linetypes found in the *icad.lin* or files. (Click **Browse** to choose a different *.lin* file, including those from AutoCAD.)




TIPS progeCAD stores both simple and complex linetypes in *icad.lin* or *icadiso.lin*.

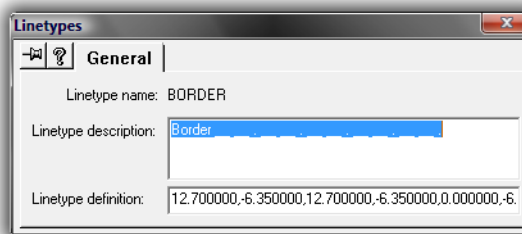
In addition, progeCAD imports linetypes from AutoCAD.

3. Select a linetype (by name, not by pattern), and then click **OK**.
4. Back in the Explorer, notice that the linetype is listed, and that you can immediately edit the name, if you so wish.

TIP You can sort the list of linetypes by clicking the headers. Click the same header to sort in reverse order:

- **Linetype name** sorts them in alphabetical order by name.
 - **Linetype Description** in order of description.
 - But clicking **Linetype Appearance** doesn't sort.
-

5. Whether imported or newly created by you, you can edit the linetype: Right-click the name of the linetype, and then select **Properties**. (Alternatively, click the  **Properties** button on Explorer's toolbar.)



6. Notice that you can change the description and the definition. The change takes effect when you exit this Linetypes dialog box by clicking the **x** button in the upper right corner.

(The pushpin button allows this dialog box to hang around, but I fail to see the point, since changes don't take place until you dismiss the dialog box.)

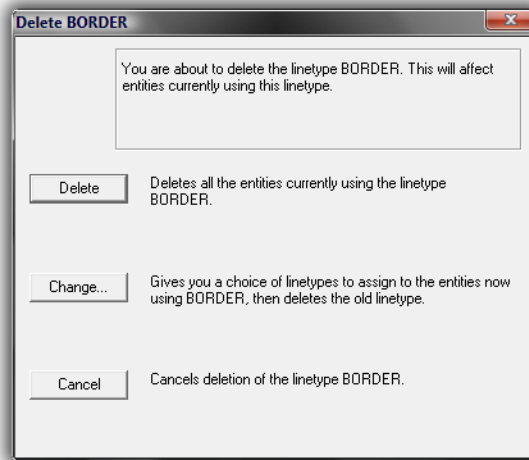
It might be fixed in your copy of progeCAD, but I could not get the Properties dialog box to accept changes to complex linetypes.

Deleting Linetype Definitions

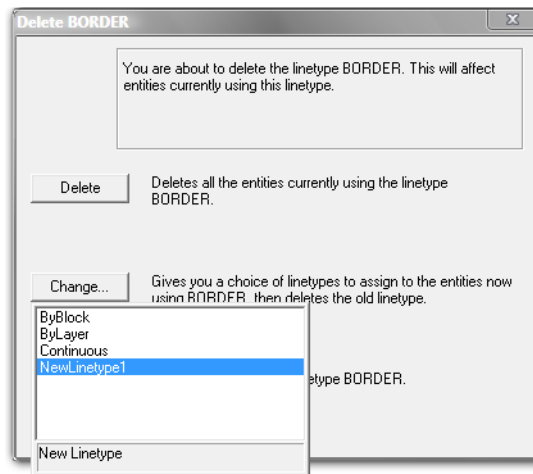
It's an unpleasant task, but sometimes you gotta do it: erase linetypes.

1. In Explorer, select the linetype to delete.
2. Right-click, and select **Delete**. (Alternatively, press the **Del** key on your keyboard. Or, click the angry red **X** button on the toolbar.)

Notice that progeCAD flashes an impressive looking dialog box warning you of your action. progeCAD erases the linetype definition along with **all objects** drawn with that definition, which I think is somewhat excessive.



The better option is to click **Change**. This not only saves the entities from eternal destruction, but allows you to substitute a different linetype.



3. Double-click one of the linetypes listed by the **Change** button. Back in Explorer, the selected linetype is gone, and entities have taken on the other linetype.

At the Command Prompt

Follow these steps to create a new linetype on-the-fly, using the **Linetype** command:

1. Start progeCAD, and then enter the **Linetype Create** command:

Command: **linetype**

Linetype: ? to list/Create/Load/Set: **c**

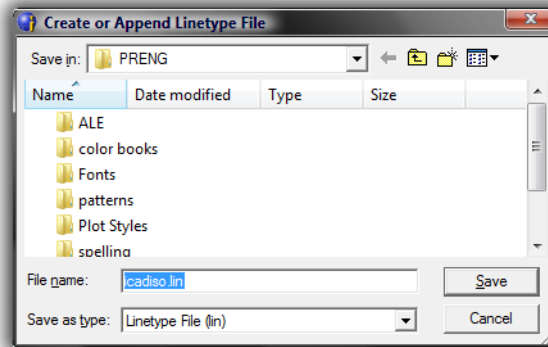
2. Give a name to the linetype, which can be as long as 31 characters.

Name for new linetype: **dit-dah**

Unlike creating a custom hatch pattern on-the-fly, progeCAD actually stores the new linetype in a *.lin* file, letting you reuse it later.

3. At this point, progeCAD pops up the Create or Append Linetype dialog box. This lets you place the custom linetype in a new *.lin* file, or else have progeCAD append the linetype description to the *icad.lin*.

I find it easiest to keep all linetypes in one file, so I recommend accepting *icad.lin* — or *icadiso.lin* if I tend to work with ISO (international standard) linetypes.



4. After clicking the **Save** button to dismiss the dialog box, progeCAD checks:

One moment... Checking existing linetypes for "dit-dah".

If two linetypes have the same name, progeCAD would only ever read the first one it comes across. If you accidentally (or otherwise) enter a linetype name that already exists — such as Dashed — progeCAD warns:

One moment... Checking existing linetypes for "dashed".

DASHED already exists. Current definition is:

DASHED

0.50,-0.250

Overwrite? <N>:

In this case, press **ENTER** and then try naming again.

5. Next, describe the linetype with any words you want up to 47 characters long.

Linetype description: . _ . _ . _ . _ . _

A good descriptive text would be the pattern you plan to create, using dots, underlines, and spaces.

6. Finally! You get to define the linetype pattern. But, what's this A? The letter **A** forces the linetype to *align* between two endpoints. That's what causes the linetypes start and stop with a dash, adjusted to fit. The **A** could also stand for "actually" because, actually, I don't have a choice when I create a linetype on-the-fly: progeCAD forces the letter A on me.

Linetype definition (positive numbers for lines, negative for spaces):
A,

Type the codes after the A, as follows:

A, .25,-.10,-.1

I could go on for a total of 78 characters but I won't.

7. I press **ENTER** to end linetype definition, and I'm done.

Linetype "dit-dah" was defined in C:\progeCAD\lacad.lin.
Linetype: ? to list/Create/Load/Set: (Press Enter.)

Well, not quite. I still need to test the pattern. By the way, new linetypes are added to the *end* of the *icad.lin* or *icadiso.lin* file.

Testing the New Linetype

It is important to always test a new customization creation. As simple as they are, linetypes are no exception. Test the Dit-Dah pattern, as follows:

1. Use the **Linetype Load** command to load the pattern into drawing:

```
: linetype  
Linetype: ? to list/Create/Load/Set: L  
Enter linetype to load: dit-dah
```

2. Up pops the Select Linetype File dialog box. Select *icad.lin*, and then click **Open**. progeCAD confirms:

Linetype DIT-DAH loaded.

3. Use the **Set** option to set the linetype, as follows:

```
~/Create/Load/Set: s  
New entity linetype (or ?) <BYLAYER>:
```

4. Here you can type either the name of a loaded linetype (such as "dit-dah") or type ? to see which linetypes are already loaded.

5. This time, get serious and set the current linetype to "dit-dah":

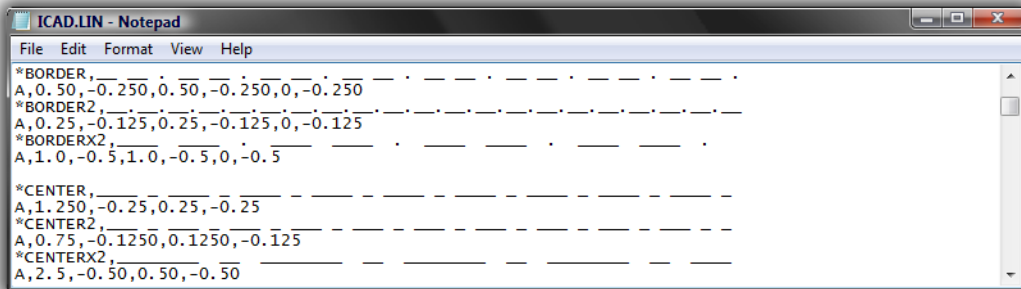
```
~/Create/Load/Set: s  
New entity linetype (or ?) <BYLAYER>: dit-dah  
~/Create/Load/Set: (Press ENTER.)
```

6. Now, draw a line, and appreciate the linetype it is drawn with. Your debugging session is over.

Creating Linetypes with the Text Editor

You can edit the *icad.lin* linetype file directly to create custom linetypes. Here's how:

1. Start a text editor (not a word processor), such as NotePad.
2. Load the *icad.lin* file from the *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG* folder. (Replace “login” with the login name you use with Windows.)



3. When you scroll down to the end of the file, you see the Dit-Dah pattern you defined as per the earlier tutorial.
4. You can modify an existing linetype, or add a new linetype. The process is exactly the same as when you did it within progeCAD, with two exceptions: (1) progeCAD isn't there to prompt you; and (2) You don't need to use the “A” prefix.
5. Save the *.lin* file with the same name (*icad.lin*) or a new name, then test it within progeCAD.

The Linetype Format

The linetype definition consists of two lines of text:

Line 1: Header

Line 1 is the header, such as “*dit-dah, . _ . _ . _”, where:

- * (asterisk) — indicates the start of a new linetype definition. DIT-DAH Name of the linetype.
- , (comma) — separates the name from the description.
- . _ _ . _ _ — describes the linetype (to a maximum of 47 characters), which is displayed by the **Linetype ?** command.

Line 2: Data

Line 2 is the data, such as “A, .25,-.1,0,-.1”, where:

- A** The “A” is the optional alignment flag, which forces progeCAD to start and end the linetype with a line.
- .25** The first number is the length of a dash when **LtScale** = 1.0; every linetype data line must begin with a dash.
- .1** Numbers with negative signs specify the length of a gap when **LtScale** = 1.0; every linetype data line must follow the initial dash with a gap.
- 0** Zeros draw dots.

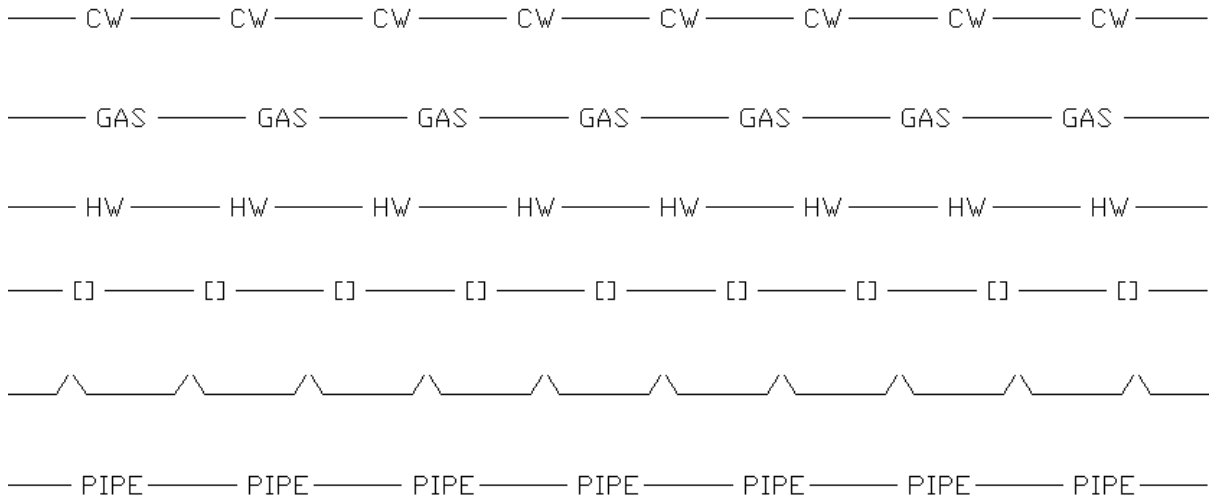
You can use a semicolon (;) to prefix any line as a comment line. Anything after the semicolon is ignored by progeCAD.

Complex (2D) Linetypes

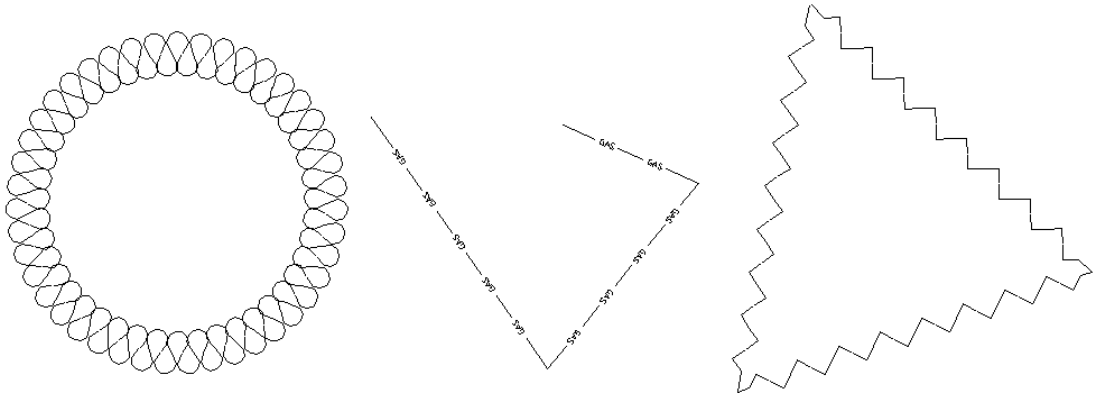
“Complex” linetypes are 2D: they can wiggle back and forth (within limits) and include text characters. Truth be told, that’s all they are: text — or, more accurately, *shapes*. See Chapter 7 for full information on coding shapes.

The complex linetype is a mixture of 2D shapes and the 1D linetype codes — the dash, gap, and dot you learned of earlier. The 2D shapes can be a combination of: (1) text characters from an *.shx font* file; and/or (2) shapes from an *.shx shape* file.

progeCAD 5 supports complex linetypes, storing their definition in the *icad.lin* or *icadiso.lin* file. Shown below are samples.



Notice that the “square” linetype uses square brackets to create the box effect: [and]. The “zig zag” linetype uses the slash and backslash characters: / and \. You might be able to think of other ASCII characters to use in a creative manner, such as the small letter o and the tilde, ~.



TIPS progeCAD is *unable* to compile source *.shp* files into *.shx*, which are needed to create custom complex linetypes. This means that you need AutoCAD to access the **Compile** command, which performs the compilation.

If all you need is text in a complex linetype, then you can use any *.shx* font file.

Embedding Text

Two of the linetypes included with progeCAD, the gas line and the hot water supply, simply combine a dash and gap with the letters GAS and HW from the Standard text style (as defined by the *txt.shx* font file). Here is the code for hot water:

```
*HOT_WATER, Hot Water ----HW----HW----HW----HW----HW----HW--
A,1.0,-.25,["HW",STANDARD,S=-.2,R=0.0,X=-0.1,Y=-0.1]-.40
```

Much of this looks familiar, with the exception of the stuff between the square brackets, shown in **boldface**. That allows the embedding of text in a linetype, and here’s what it means:

Text - "HW"

"HW" prints these letters between the dashes.

Text Style - STANDARD

STANDARD applies this text style to the text. This is optional; when missing, progeCAD uses the current text style, stored in system variable **TextStyle**.

Text Scale - S=.2

S=.2 is the text scale factor. It means one of two things:

- When the **height** defined by the text style is 0 (as is often the case), then **S** defines the height; in this case, the text is drawn 0.2 units tall).
- When the text style height is *not* 0, then this number multiplies the text style's height; in this case, the text is drawn at 0.2 times (or 20%) of the height defined in the text style.

Text Rotation - R=0.0

R=0.0 rotates the text relative to the direction of the line; in this case, 0.0 means there is no rotation. The default measurement is degrees; other forms of angular measurement are:

- **r** for radian, such as $R=1.2r$ (there are 2π radian in a circle).
- **g** for grad, such as $R=150g$ (there are 400g in a circle).

This parameter is optional and can be left out. In that case, progeCAD assumes zero degrees.

Absolute - A=0.0

(Optional) **A=0.0** rotates the text relative to the x-axis (the "A" is short for *absolute*). This ensures the text is drawn always oriented in the same direction, no matter the angle of the line. By the way, the rotation is always performed within the text baseline and capital height. That's so the text isn't rotated way off near the orbit of Pluto.

X and Y Offset - X=-0.1 and Y=-0.1

X=-0.1 shifts the text in the x-direction from the linetype definition vertex, which helps center the text in the line. **Y=-0.1** shifts the text in the y-direction from the linetype definition vertex. In both cases, the units are in the linetype scale factor, which is stored in system variable **LtScale**.

Summing up, you can create a text-based linetype with a single parameters, such as ["HW"], or you can exercise fine control over the font, size, rotation, and position with the six parameters listed above. progeCAD can work with any *.shx* font file you have on your computer.

Embedding Shapes

The other possibility is to embed a *shape* from an *.shx* file. This is not as easy as embedding text, because progeCAD does not (yet) compile source shape files. Here is how a complex linetype definition would look with an embedded shape, using shapes found in the *ltypeshp.shx* file:

```
*SHAPES,Shape SSS  
A,.0001,[SSS,ltypeshp.shx,x=-.01,s=-.02],-.0001
```

Let's look at the relevant parts that differ from embedding text, as highlighted in **boldface** above:

Shape Name - SSS

SSS is the name of the shape. When progeCAD cannot find the shape, it draws a ? instead.

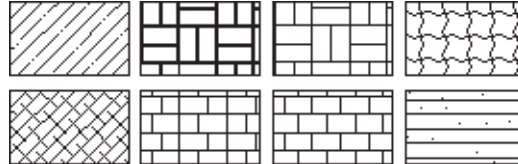
Shape File - ltypeshp.shx

ltypeshp.shx is the name of the file holding the shapes.

TIP Complex linetypes must start and end with a dash or a gap. You cannot write a linetype definition that consists of shapes only.

Making Hatch Patterns

Despite their seeming complexity, hatch patterns consist of three basic elements: dashes, gaps, and dots. To create repeating patterns, the basic definition is offset by distance and an angle.



Even though progeCAD comes with 560 patterns, your office drafting standard may require a specific pattern. In this chapter, we look at how to create hatch patterns, and edit existing patterns.

In This Chapter

- Where do hatch patterns come from?
- Creating custom hatch patterns.
- Using the Hatch and BHatch commands.
- Understanding the icad.pat file.
- Tips on creating pattern codes.
- Adding custom patterns to the palette.
- Creating hatch patterns.
- Creating slides.
- Catalog of hatch pattern samples.

Where Do Hatch Patterns Come From?

Hatch patterns are defined in files external to progeCAD called *icad.pat*, *icadiso.pat*, and *extras.pat*. You can have many pattern files, each with an extension of *.pat*. It is easier, however, to keep all patterns in a few files.

progeCAD stores its patterns files in the *C:\Program Files\progeSOFT\progeCAD 2009 Pro ENG\User Data Cache\patterns* folder. Unlike linetypes, the pattern file is loaded automatically the first time you use the **Hatch** or **BHatch** commands (short for “boundary hatch”).

In addition, progeCAD uses *.sld* slide files to show pattern samples in the Pattern tab of the Boundary Hatch dialog box — one slide file per pattern sample.

The hatching patterns consist only of lines, line segments (dashes), dots, and gaps; progeCAD cannot create hatch patterns made of circles and other nonlinear objects. progeCAD can solid-fill areas in any of its 255 colors.

How Hatch Patterns Work

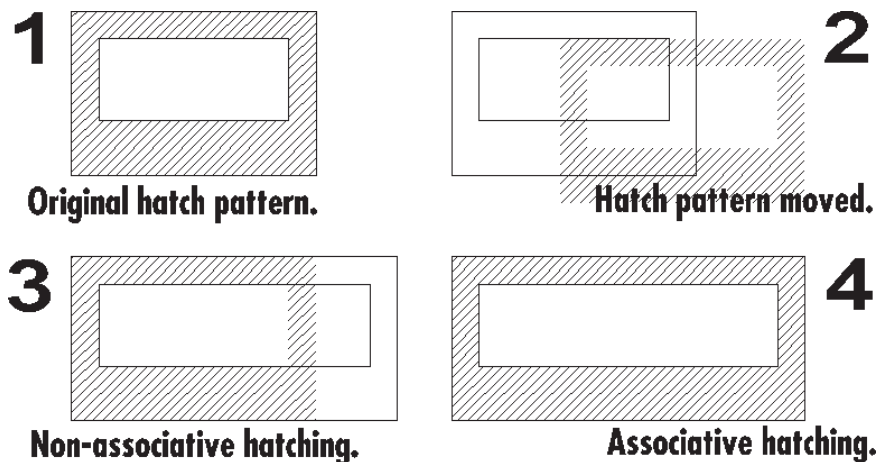
The **Hatch** command creates hatch patterns at the command line; **BHatch** displays a dialog box to do the same thing.

When you apply a hatch to an area, progeCAD generates an repeating pattern of parallel lines and gaps from the definition in the *.pat* file. The pattern comes to a stop when it reaches a boundary; if progeCAD cannot detect a boundary, it refuses to place the pattern.

Once the hatch is in place, you can use the Move command to move the hatch pattern elsewhere in the drawing, if you so chose.

progeCAD can create non-associative and associative hatch patterns through the Pattern Properties tab of the Boundary Hatch dialog box:

- *Non-associative* means the shape of the pattern’s area is fixed; when you change the boundary, the pattern does not change. This is useful when you want the pattern to remain fixed.
- *Associative* hatching means the pattern’s shape updates as you change the boundary.



progeCAD treats both kinds of hatch pattern as a block. You can use the **Explode** command to explode the block into its constituent lines. progeCAD cannot edit a hatch pattern; the workaround is to erase, and reapply.

progeCAD has several system variables that report the most-recent setting of hatch pattern parameters:

HpName	Specifies the name of the current hatch pattern (default = ANSI31).
HpBound	Determines the boundary entity created by the BHatch commands: 0 = regions; 1 = polylines
HpDrawOrder	Controls the display order of hatches: 0 Hatches are not assigned draw order. 1 Hatches are displayed behind of all other entities. 2 Hatches are displayed in front of all other entities. 3 Hatches are displayed behind their boundaries. 4 Hatches are displayed in front of their boundaries.
HpStyle	Determines the hatch pattern style: 0 = standard pattern style; 1 = outer pattern
HpScale	Specifies the current scale factor (default = 1.0).
HpAng	Specifies the current angle of the hatch pattern in degrees (default = 0).
HpDouble	Determines whether the hatch is applied a second time at 90 degrees.
HpSpace	Specifies the spacing between hatch pattern lines (default = 1.0 units).
SnapAng	Specifies the rotation angle of the hatch pattern in degrees (default = 0).
SnapBase	Specifies the x,y-coordinates of the origin for the hatch pattern (default = 0,0).

The last two system variables let you control where the hatch pattern begins. Normally, the pattern assumes an origin of (0,0) and an angle of 0 degrees. But if you need to precisely control the placement of the pattern, change the values of **SnapAng** and **SnapBase**, as required.

Creating Custom Hatch Patterns

progeCAD has two ways to create a custom hatch pattern: (1) simple patterns at the 'Command:' command prompt or dialog box; and (2) edit the *.pat* files with a text editor. We look at both methods in this chapter.

To create simple hatch patterns, use the Hatch or BHatch commands. progeCAD does not, unfortunately, save the fruit of your labors (unlike when you create a custom linetype with Linetype.) For this reason, think of the first method of creating custom hatch pattern on-the-fly.

Hatch Command

Your options for creating a hatch on-the-fly are limited to simple patterns. Here's how, using the **Hatch** command:

1. Start progeCAD.
2. Enter the **Hatch** command:
Command: **hatch**
3. Select the **&** option:
Hatch: ? to list patterns/SEttings/& for lines/Preview/STYLE/<Pattern name> <>: **&**
4. Specify the three parameters for the custom hatch pattern.

a. First, the *angle*.

Proceed/Angle for pattern <0>: **45**

TIP The hatch angle is measured from the setting of system variable **SnapAng** (0 degrees, by default, which is in the direction of the positive x-axis). When **SnapAng** is set to something other than 0, the angle you specify here is added to the value stored in **SnapAng**.

b. Second, the *spacing* between parallel lines

Space between standard pattern lines <1.0000>: **2**

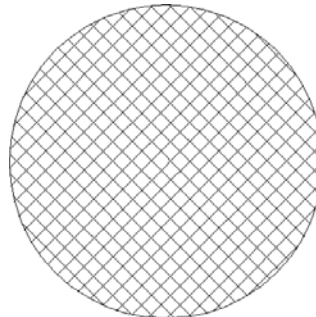
c. Third, decide if you want the pattern *crosshatched*. That means a second pattern is applied at 90 degrees to the first pattern.

Cross-hatch area? <No>: **y**

5. Finally, you select the object or boundary to hatch:

ENTER to apply hatch/<Starting point>: (Press Enter to apply the pattern.)

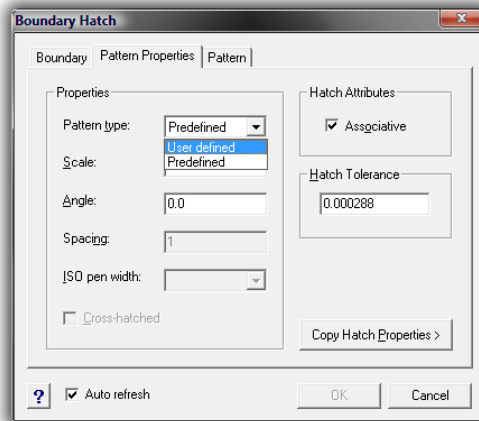
progeCAD draws the pattern, but — as mentioned earlier — your custom hatch isn't saved to the *.pat* file.



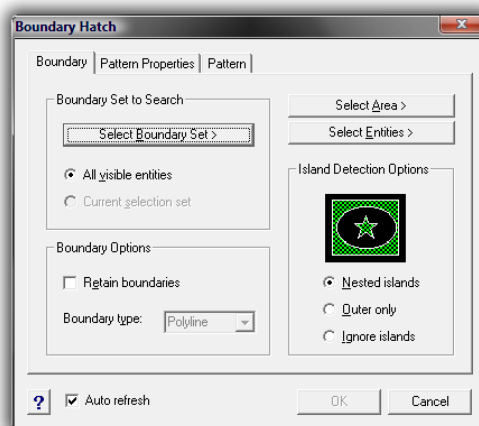
BHatch Command

To create a custom hatch pattern with the **BHatch** command is a bit different; it's more like filling out a form:

1. From the **Draw** menu, select **Hatch**. (Or enter the **BHatch** command.)
2. When the Boundary Hatch dialog box appears, select the **Pattern Properties** tab.
3. From the **Pattern Type** drop list, select **User Defined**.



4. progeCAD allows you to type values for **Angle**, **Spacing**, and **Cross-hatched**. Enter a value for each of these.
5. Select the **Boundary** tab.
6. Click the **Select Area** button, and then select the area you want hatched.



7. Press **ENTER** to return to the dialog box, and then click **OK**. progeCAD applies the custom pattern.

Caution: Whether you use **Hatch** or **BHatch**, in neither case is the hatch pattern you created saved in a *.pat* file — unlike linetypes.

Understanding the .pat Format

Let's now dig into the contents of the *icad.pat* file to get a better understanding of how a pattern is constructed.

1. Start a text editor (not a word processor), such as Notepad.
2. Open the *icad.pat* file from the *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\patterns* folder. (Replace “login” with the login name you use to access the computer.)
3. Scroll down a bit, and take a look at the seemingly-incomprehensible series of numbers and punctuation contained by this file. I've reproduced the first dozen lines here:

```
*SOLID, Solid fill
45,0,0,0,0.1
*ANGLE, Angular Steel
0, 0,0, 0,,275, .2,-.075
90, 0,0, 0,,275, .2,-.075
*ANSI31, ANSI
45, 0,0, 0,0.125
```

Comment and Header Lines

The definition of a hatch pattern consists of two or more lines of text. The first line is called the *header*, such as **SOLID, Solid fill*.

Comment - ;

The semicolon (;) indicates a comment line, such as ; Note: Dummy pattern description used for 'Solid fill'. That lets you include notes to yourself that are ignored by progeCAD.

Start of Definition - *

The asterisk (*) is important because it signals to progeCAD the start of a new hatch pattern definition.

Pattern Name

Following the asterisk comes the name for the hatch pattern, such as *SOLID*. The name must be unique in the file. If it isn't, progeCAD uses the first pattern it finds by that name.

The comma following the name merely separates the name from the description. The comma is optional; it doesn't have to be there: a space works just as well.

Description

The text following the pattern name is the description displayed by the **Hatch ?** command, such as “Solid fill.” This description is also optional, but highly recommended. You are limited to a maximum of 80 characters for the name, comma, and the description. If you need more room for the description, use comment lines, such as:

```
; Note: Dummy pattern description used for 'Solid fill'.
*SOLID, Solid fill
```

The Hatch Data

With the comment lines and the header line out of the way, let's get down to the nitty-gritty hatch pattern data and how it is coded. Lines 2 and following are the data, such as:

```
0, 0,0, 0,.275, .2,-.075 90, 0,0, 0,.275, .2,-.075
```

Every line of data uses the same format:

```
angle, xOrigin, yOrigin, xOffset, yOffset [, dash1, ...]
```

angle

Angle is the angle at which this line of hatch pattern data is displayed. The “0” means the hatch line is drawn horizontally; a “90” means the line is drawn vertically, and so on.

A comma (,) separates the numbers.

xOrigin and *yOrigin*

The *xOrigin* specifies that the first line of the hatch pattern passes through this x-coordinate. The value of the *yOrigin* means that the first line of the hatch pattern passes through this y-coordinate.

xOffset and *yOffset*

The *xOffset* specifies the distance between line segments, a.k.a. the *gap* distance. You use this parameter only to specify the offset for vertical or diagonal lines (To specify the distance between dashes, use the **dash1** parameter.) In most hatch patterns, *xOffset* has a value of 0.0. Even though this parameter is rarely used, it is not optional.

The *yOffset* is the vertical distance between repeating lines; this parameter is used by every hatch pattern.

dash1,...

dash1 defines the dashes in the hatch pattern line (the code is the same as for linetypes):

- A positive number, such as 0.25, is the length of the dash.
- A 0 draws a dot.
- A negative number, such as -0.25, draws a gap.

TIP The dot drawn by the hatch pattern may create a problem when it comes time to plot. If you find that the dots in a hatch pattern are not printed, use a very short line segment, such as 0.01, instead of a 0.

When you are finished editing a pattern, save the *.pat* file.

Tips on Creating Pattern Codes

Some miscellaneous comments on hatch pattern coding:

Tip 1: Each line of code applies to a single pattern segment; the two lines of data (above) represent a hatch pattern with two lines.

Tip 2: Hatch pattern lines are drawn infinitely long. What this means is that progeCAD draws the line as long as necessary, as long as it reaches a boundary. progeCAD will not draw the hatch pattern unless it does find a boundary.

Tip 3: At the very least, each line of pattern code must include the **angle**, **x-** and **y-origin**, and the **x-** and **y-offset**. This draws a continuous line.

Tip 4: The **dash1** parameter(s) is optional but when used draws a line with the dash-gap-dot pattern.

Tip 5: There is no practical limit to the number of data lines for a hatch pattern definition. Very complex patterns can take dozens and dozens of lines of code. But be careful: a complex hatch pattern takes a long time to draw on slower computers. For this reason, place hatch patterns on their own layer in a drawing, then freeze that layer. Thaw the layer when you need to see the pattern or plot the drawing.

Tip 6: To change the angle of a hatch pattern upon placing it in the drawing, you've got a couple of options:

- Specify the angle during the **Hatch** and **BHatch** commands.
- Set the angle in system variable **SnapAng**. The effect of **SnapAng** on the hatch pattern angle is additive: if the hatch pattern defines the lines drawn at 45 degrees and **SnapAng** is 20 degrees, then progeCAD draws the hatch lines at 65 degrees.

For example:

```
: snapang  
New current angle for SNAPANG <0>: 20
```

Tip 7: The **x-offset** and **y-offset** parameters are unaffected by the angle parameter, because **x-offset** is always in the direction of the line and **y-offset** is always perpendicular (90 degrees) to the line.

Tip 8: For whatever reason, progeCAD does not make it easy to change the origin of the hatch pattern, which is important for accurate placement of the patterns or lining the pattern up with another pattern. To change the x,y-origin of a hatch pattern upon placing it in the drawing, use system variable **SnapBase**. The effect of **SnapBase** on the hatch pattern origin is additive: if the hatch pattern specifies that the lines start at 0.1,0.11 and **SnapBase** is 5,5, then progeCAD starts the hatch at 5.1,5.1.

Tip 9: If you are uncomfortable using system variables, then the **Snap** command provides the same opportunity via the **Rotate** option:

```
: snap
Snap is off (x and y = 0.5000): ON/Rotate/Style/Aspect/<Snap spacing>: r
Base point for snap grid <0.0000,0.0000>: 1,1
Rotation angle <0>: 45
```

Tip 10: You cannot specify a weight (or linewidth) for a hatch pattern line. The workaround is to define two or more very closely spaced lines, such as:

```
*Thick_Line, Closely spaced lines
0, 0,0, 0.,25 0, 0.,01, 0.,25 0, 0.,02, 0.,25
```

Tip 11: You cannot specify arcs, circles, and other round elements in a hatch pattern file. Everything consists of straight lines and dots. To simulate circular elements, use a series of very short dashes.

Tip 12: To draw dash and gap segments at an angle, use the sine of the angle in degrees, like this:

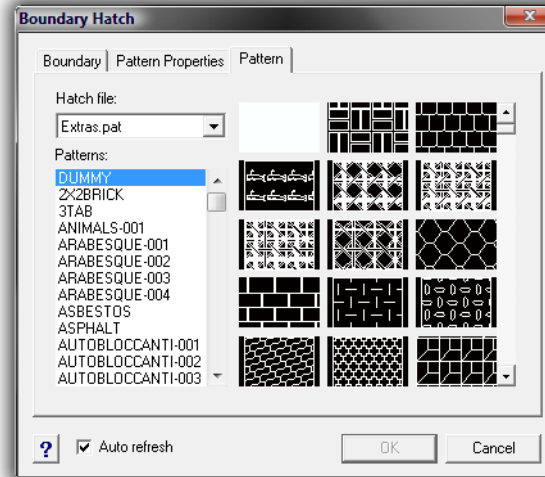
<u>Angle</u>	<u>Dash length (sine)</u>
0	0
30	0.433
45	0.707
60	0.866
90	1.0

Tip 13: It's a lot easier for someone else (or you, six months from now) to read your hatch pattern code if you use tabs and spaces to format the code into nice columns.

Adding Custom Patterns to the Palette

Finally, let's see how to add samples of your custom-made hatch pattern(s) to the BHatch command's dialog box. You cannot, unfortunately, expect your new pattern definitions to appear automatically.

Let's take a look at the problem: I added a dummy test hatch pattern (called "Dummy") to one of the *.pat* files. When I started the **BHatch** command, the name "Dummy" appears, but the sample preview is blank. See figure below. The problem: how do we make the pattern appear in the preview area.



There are two parts to solving the problem: (1) create a sample of the hatch pattern; and (2) save the sample as a slide file in the *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\patterns* folder. (Replace "login" with the login name you use to access the computer.)

Creating a Sample Hatch Pattern

Here are the steps I took to create a sample of the hatch pattern:

1. Start progeCAD.
2. The **Boundary Hatch** dialog box uses an icon image with an aspect ratio of 1.64:1. Set up a paper space viewport, because that restricts the **MSlide** command to taking a "snapshot" of the viewport, not the entire drawing screen. I typed the following commands to set up the screen:

```
Command: tilemode 0  
Command: mview 0,0 1.64,1  
Command: zoom e  
Command: rectang 0,0 1.64,1
```

The **Tilemode 0** command turns off tilemode (to allow paper space viewports).

The **MView 0,0 1.64,1** command creates a viewport with the 1.64:1 aspect ratio.

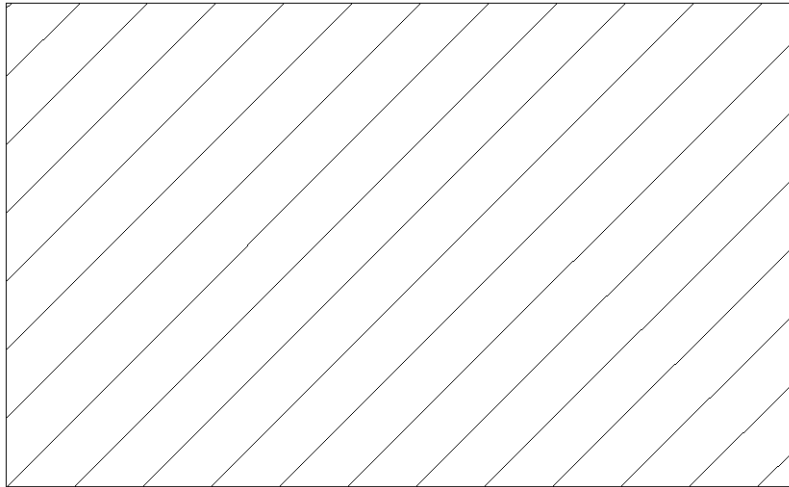
The **Zoom E** command makes the viewport fill the drawing screen to maximum size.

And the **Rectang 0,0 1.64,1** command draws a boundary for the hatch pattern.

3. I can now use the **BHatch** (or **Hatch**) command to hatch the rectangle.

Command: **bhatch**

If at all possible, I use a scale of 1.0; this lets me see the hatch pattern's size relative to other hatch patterns. I only use a larger or smaller scale if I cannot see a representative sample of the pattern.



Creating the Slide

The second step is to convert the hatch sample into a *.sld* file.

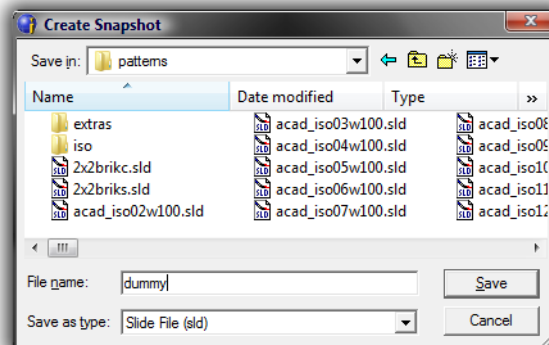
1. Before making the slide, I switched back to model space:

Command: **mpace**

2. Then, I created the slide file:

Command: **mslide**

This command displays the **Create Snapshot** dialog box. Change the folder to *C:\Users\login\AppData\Roaming\progeSOFT\progeCAD 2009\R9\PRENG\patterns*. Type the same name as the hatch pattern, such as *dummy.sld*. Then click the **Save** button.



3. (*Optional*) When I have more than one hatch pattern of which to make slides, I switch back to paper space before employing the **Erase** command to remove the old hatch pattern:

Command: **pspace**

Command: **erase**

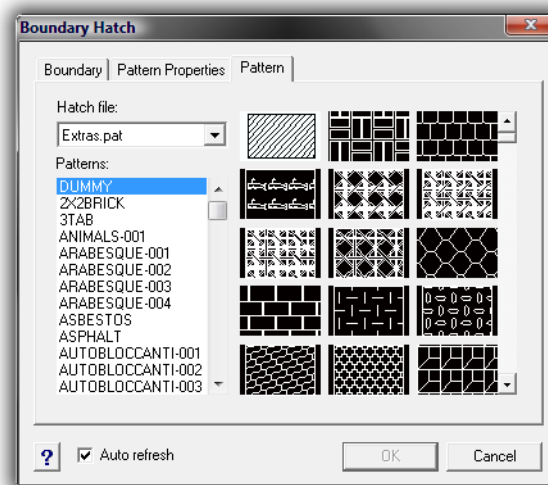
After erasing the pattern, I applied a new pattern with **BHatch**. Again, I switched back to model space, and retook the slide:

Command: **bhatch**

Command: **mSPACE**

Command: **mslide**

4. As always when customizing progeCAD, it is import to test my work, to make sure the slide appears. I start **BHatch**, and then select the **Pattern** tab. When I select the name of the custom hatch pattern, the slide image appears!



TIP If the pattern fails to appear, I have found it helps to exit progeCAD, restart it, and then check again with **BHatch**. This forces progeCAD to reload the *.sld* files that hold the pattern images.

Creating Shapes & Fonts

progeCAD uses *.shp* and *.shx* files for fonts, complex linetypes, and GDT symbols. You can create your own *.shp* files, which is the subject of this chapter. progeCAD lacks, however, the **Compile** command, and cannot compile *.shp* files. It does support shape files as simple blocks, and hence includes the **Load** and **Shape** commands.

progeCAD can display fonts in TrueType (*.tff*) and AutoCAD formats (*.shx*). If a drawing displays fonts incorrectly, the problem lies with progeCAD not finding the location of the font files. Use the **Tools | Options | Paths** command to add paths to the Fonts item.

In This Chapter

- Shapes with fonts, complex linetypes, and GDT symbols.
- About shape files.
- Font compatibility with AutoCAD.
- Using shapes in drawings.
- Understanding the shape file format.

Fonts, Complex Linetypes, and GDT Symbols

progeCAD uses shapes in three areas: for fonts, complex linetypes, and GDT symbols.

Fonts

Fonts were originally coded as highly-efficient shapes in the early days of CAD, because text was one of the slowest parts of the drawing display. Shown below are the worst and best of *.shp*-based fonts:

Shp Fonts
Shp Fonts

By being constructed of mostly straight line, the text displayed faster on slow computers.

The drawback to shapes, however, is that they are not well-suited to defining the complex curves that truly represent fonts, nor can they be filled. For this reason, progeCAD also supports *.ttf* TrueType fonts, which are commonly provided with the Windows operating system.

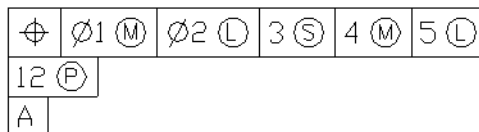
Font definition files are found in the `\progeCAD\fonts` folder. Fonts are loaded with the **Style** command, and then placed with the **Text** and **MText** commands. The Text Style dialog box previews TrueType fonts, but not *.shx* fonts.

Complex Linetypes

Complex linetypes use shapes to produce the squiggles and text. Linetypes are defined by the *icad.lin* and *icadiso.lin* files, found in the `\progeCAD` folder. Complex linetypes are loaded and placed with the **Linetype** command. See Chapter 5 for details.

GDT Symbols

GDT symbols (geometric dimensioning and tolerancing) is the other area where shapes are still used.



They are defined by shapes in the *ic-gdt.shx* file, found in the `\progeCAD\fonts` folder. The symbols are placed with the **Tolerance** command.

About Shape Files

There are two kinds of shape files, *.shp* and *.shx*. The differences between them are as follows:

- ***.shp*** are shape source files. When you write or edit a shape or font, you work with the *.shp* file. A portion of a typical *.shp* file looks like this:

```
*130,6,TRACK1
014,002,01C,001,01C,0
```

progeCAD has some *.shp* source files in the `\progeCAD\fonts` folder.

- ***.shx*** are compiled shape files. ***progeCAD is incapable of compiling .shp files into .shx format. You must use AutoCAD to compile the shape files you create using the information in this chapter.*** Normally, you cannot edit *.shx* files, unless you have access to a shape decompiler program written by third parties. (A search in Google for “shx decompilers” comes up with several products.)

Font Compatibility with AutoCAD

progeCAD uses the *icad.fmp* file to substitute its own *.shx* fonts for those found in AutoCAD drawings; *.fmp* is short for “font map.”

<u>AutoCAD Font</u>	<u>progeCAD Equivalent</u>	<u>AutoCAD Font</u>	<u>progeCAD Equivalent</u>
Bigfont	ic-txt.shx	ScriptC	ic-scrpc.shx
Complex	ic-comp.shx	ScriptS	ic-scrps.shx
GDT	ic-gdt.shx	Simplex	ic-simp.shx
GothicE	ic-gothe.shx	SyAstro	ic-txt.shx
GothicG	ic-gothg.shx	SyMap	ic-txt.shx
GothicI	ic-gothi.shx	SyMath	ic-txt.shx
GreekC	ic-grekc.shx	SyMeteo	ic-txt.shx
GreekS	ic-greks.shx	SyMusic	ic-txt.shx
Italic	ic-ital.shx	Txt	ic-txt.shx
ItalicC	ic-italc.shx	IsocP.shx	ic-isop1.shx
ItalicT	ic-italt.shx	IsocP2.shx	ic-isop2.shx
MonoTxt	ic-mono.shx	IsocP3.shx	ic-isop3.shx
RomanC	ic-romnc.shx	IsocT.shx	ic-isot1.shx
RomanD	ic-romnd.shx	IsocT2.shx	ic-isot2.shx
romanS	ic-romns.shx	IsocT3.shx	ic-isot3.shx
RomanT	ic-romnt.shx		

Using Shapes in Drawings

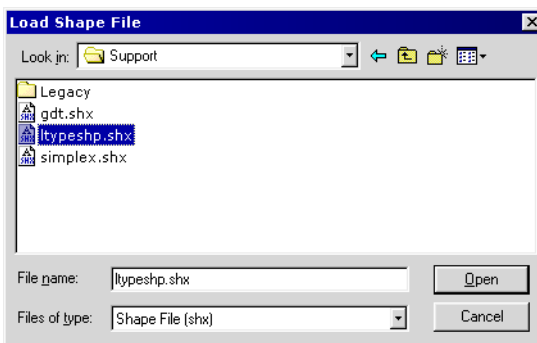
progeCAD has the ability to place shapes in drawings. To do this takes two steps: (1) load the shape file into the drawing with the **Load** command; and (2) place the shape with the **Shape** command. The problem is that progeCAD doesn't include any useful shape files. When you use the **Load** command, it opens to progeCAD's `\fonts` folder; but font `.shx` files are not shape `.shx` fonts. When you select a font `.shx` file, progeCAD complains, "File is not shape file."

If you have an older version of AutoCAD around, it included some sample `.shx` files. Even if you have a recent release, go on over to AutoCAD's `\support` folder and load the `ltypeshp.shx` file. It'll work. (Interestingly enough, the `gdt.shx` file is considered font file.)

1. Load the shape file into progeCAD, as follows:

```
: load
```

2. In the Load Shape File dialog box, use the **Look in** listbox to navigate to AutoCAD's `\support` folder.



Select the `ltypeshp.shx` file, and then click **Open**. progeCAD reports, "File `ltypeshp.shx` loaded."

3. Placing a shape in the drawing is similar to placing a block:

```
: shape  
? to list/Shape to insert: ?
```

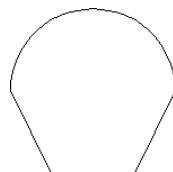
The ? option lists the names of shapes stored in the file:

```
ltypeshp.shx :  
TRACK1 ZIG BOX CIRC1 BAT
```

Repeat the **Shape** command, and this time we'll insert a shape for real:

```
: shape  
? to list/Shape to insert: bat  
Insertion point for shape: (Pick a point.)  
Scale factor for shape <1.0>: (Press ENTER.)  
Rotation angle for shape <0>: (Press ENTER.)
```

In case you were wondering, the "bat" shape is used for the insulation linetype.



The Shape File Format

Autodesk defined two formats for shape files: one for *shapes* (simple blocks) and the other for *fonts*. The difference is that fonts include a **code 0** so that the file is treated as a font definition, not a shape definition. progeCAD has the ability to load shapes and supported fonts.

A shape file typically defines one or more shapes, up to 258 shapes in total. A font file typically defines all the characters — such as A-Z, a-z, 0-9, and punctuation — for a single font. Unicode font files can have up to 32,768 definitions.

Like some other customization files, a shape definition consists of two or more lines. The first line is the header, which labels the shape, while the second (and following) lines define the shape through codes. The final code in each definition is 0, which is called the *terminator*.

Each line can be up to 128 characters in length; AutoCAD will not compile a shape file with longer lines. A single definition is limited to 2,000 bytes.

You can use blank lines to separate shape definitions, and the semicolon (;) to include comments in the file.

The general format of a shape definition a header lines, followed by one or more definition lines:

```
*shapeNumber,totalBytes,shapeName  
byte1,byte2,byte3,...,0
```

Header Fields

The following describes the fields of the shape's header description:

Definition Start - *

```
*130,6,TRACK1
```

The asterisk signals AutoCAD that the next shape definition is starting.

shapeNumber

```
*130,6,TRACK1
```

Each shape requires a unique number by which it is identified. For fonts, the number is the equivalent ASCII code, such as 65 for the letter A.

TIP The *shapeNumbers* 256, 257, and 258 are reserved for the degree, plus-or-minus, and diameter symbols.

totalBytes

```
*130,6,TRACK1
```

After defining the shape, you have to add up the number of bytes that describe the shape, including the terminator, 0. Makes no sense to me.

There is a limit of 2,000 bytes per shape definition. Unicode shape numbers count as two bytes each.

shapeName

*130,6,TRACK1

Shape names must be in all uppercase. Because names with lowercase characters are ignored, you can use them for in-line comments.

Definition Lines

The header line is followed by one or more lines that define the shape or font. This is the nitty-gritty part of shape files, and you will now see why they are rarely used anymore.

bytes

014,002,01C,001,01C,0

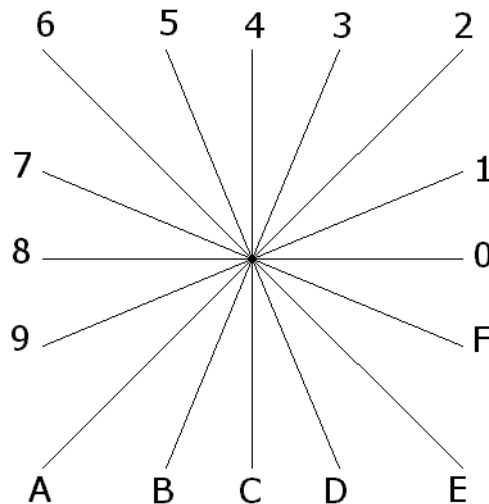
The shape is defined by “bytes,” called that because each code is a single byte (the computer measurement) in size. Bytes define vector lengths and directions, or instruction codes. They can be in decimal (base 10) or hexadecimal (base 16) format.

Definition lines are a maximum of 128 characters long (including commas), and a maximum of 2,000 bytes overall (not including commas). The last definition line ends with a 0.

TIP When the first character of a byte is a 0, the two characters following are in hexadecimal, such as **00C** (12, in decimal).

Vector Codes

Vector codes describe how the shape is drawn. They define movement (pen up) and drawing (pen down). Vector codes are limited to 16 directions, as shown by the figure:



Notice that the lengths are not radial: the diagonal vectors (such as 2 and E) are 1.414 (square root of 2) times longer than the orthogonal vectors (such as 4 and 0). Vector codes are always in hexadecimal notation, such as 02C:

- First character is always 0 to indicate that the number is in hexadecimal.
- Second character is the vector length, ranging from 1 through F (15 units).
- Third character is the direction, as noted by the figure above.

Thus, 02C would draw a line 2 units long in the -y direction (downward). By now, you can see that you need to understand hexadecimal notation.

Instruction Codes

In addition to describing direction and length, shapes use codes to provide instructions. Code numbers can be in decimal (*dec*) or hexadecimal (*hex*). Hex codes always have three digits, the first being a 0 (zero).

Notice that some codes rely on additional codes following. And, note that drawing is limited to lines, arcs, and spaces.

Hex	Dec	Description
000	0	End of shape definition.
Basic Draw and Move:		
001	1	Begin draw mode (pen down).
002	2	End draw mode (pen up).
Scaling:		
003	3	Divide vector lengths by next byte.
004	4	Multiply vector lengths by next byte.
Memory:		
005	5	Push current location onto stack.
006	6	Pop current location from stack.
Draw Subshape:		
007	7	Draw subshape number given by next byte.
Advanced Draw and Move:		
008	8	X,y displacement given by next two bytes.
009	9	Multiple x,y displacements; terminated with (0,0) code.
Arcs:		
00A	10	Octant arc defined by next two bytes.
00B	11	Fractional arc defined by next five bytes.
00C	12	Arc defined by x,y displacement and bulge.
00D	13	Multiple bulge-specified arcs.
Fonts:		
00E	14	Process next command only if vertical text code exists.

A *stack* is a specific type of memory called FILO, short for “first in, last out.” When two numbers are stored in the stack memory, the last number stored is the first one out. Think of an elevator, where the first person in is usually the last one out.

End of Shape - 0/000

Code 0 must mark the end of every shape definition. It appears at the end of the last line.

```
00C,(2,0,-127),0
```

In hex notation, 0 appears as 000.

Draw Mode - 1/001

Code 1 starts drawing mode (“pen” is down). By default, every shape definition starts with draw mode turned on.

2/002: Move Mode -

Code 2 starts move mode (“pen” is up). In the sample below, the pen is raised before moving to a new location.

```
2,8,(-36,-63),1,0
```

Reduced Scale - 3/003

Code 3 specifies the relative size of each vector. Each shape starts off at the height of one of the orthogonal vectors, such as 4. To make the shape smaller, use code 3 followed by a byte specifying the scale factor, 1 through 255. For example, the following code draws the shape half as large:

```
3,2
```

TIP Within a shape definition, the scale factor is cumulative. Using the same scale code twice multiplies the effect. For example, 3,2 followed by another 3,2 makes part of the shape four times smaller.

At the end of the shape definition, return the scale to unity so that other shapes are not affected.

Enlarged Scale - 4/004

To make the shape larger, use code 4 followed by a byte specifying the scale factor, 1 through 255. For example, the following code draws the shape twice as large:

```
4,2
```

Note that you can use the 3 and 4 codes within a shape definition to make parts of the shape larger and smaller.

Save (Push) - 5/005

Code 5 saves (*pushes*) the current x,y-coordinates to the stack memory. You then use code 6 to recall (*pop*) the coordinates for later use. The stack memory is limited to four coordinates. By the end of the shape definition, you must recall all coordinates that you saved; i.e., there must be an equal number of code 5s and 6s, as shown below:

```
2,14,8,(-8,-25),14,5,8,(6,24),1,01A,016,012,01E,02C,02B,01A,2,  
8,(8,5),1,01A,016,012,01E,02C,02B,01A,2,8,(4,-19),14,6,  
14,8,(8,-9),0
```

Recall (Pop) - 6/006

Code 6 recalls (*pops*) the most-recently saved coordinates from the stack memory.

Subshape - 7/007

Code 7 calls a subshape, which is simply another shape. Shapes can be used within other shapes, which helps reduce the tedium of coding shapes. Code 7 is followed by reference to another shape number, between 1 to 255. (Recall that all shapes within a *.shp* file are identified by number.) For example:

7,2

calls shape 2 as a subshape.

X,y Distance - 8/008

Codes 8 and 9 overcome the restriction that the vector codes (just 16 directions) place on drawing. Code 8 defines a distance using two bytes that range from -128 to 127:

8,xDistance,yDistance

The example below shows code 8 being used often:

2,14,3,2,14,8,(-21,-50),14,4,2,14,5,8,(11,25),1,8,(-7,-32),2,
8,(13,32),1,8,(-7,-32),2,8,(-6,19),1,0E0,2,8,(-15,-6),1,0E0,2,
8,(4,-6),14,6,14,3,2,14,8,(21,-32),14,4,2,0

In the first line of code above, **8,(-21,-50)** draws 21 units down (-x), and 50 units left (-y).

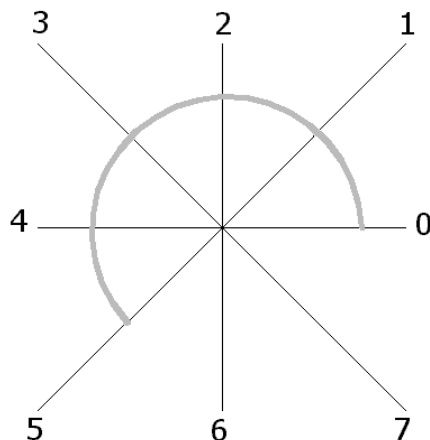
X,y Distances - 9/009

Whereas code 8 specifies a single coordinate, code 9 specifies a series of coordinates, terminated by (0,0). For example:

9,(1,2),(-3,4),(5,-6),(0,0)

Octant Arc - 10/00A

Code 10 defines an *octant arc*, which is an arc whose angle is limited to multiples of 45 degrees, as shown in the following figure. The arc always starts at position 0, and move counterclockwise.



The arc is specified by the following bytes:

```
10,radius,- 0 startingOctant octantSpan
```

- **10** specifies an octant arc.
- *radius* is a value between 1 and 255.
- Negative sign changes the direction of the arc to clockwise; leave it out for counterclockwise direction.
- **0** specifies the following characters are hexadecimal.
- *startingOctant* specifies where the arc starts; the value ranges between 0 and 7).
- *octantSpan* specifies how hard the arc travels, again a number between 0 through 7.

TIPS When *octantSpan* is **0**, the shape draws a circle.

The octant arc code usually uses parentheses to make itself clearer, such as:
10,(25,-040)

Fractional Arc - 11/ 00B

Code 11 is more useful because it draws arcs that don't end and start at octant angles. Its specification requires, however, five bytes:

```
11,startOffset,endOffset,highRadius,radius,- 0 startingOctant octantSpan
```

- **11** defines the fractional arc.
- *startOffset* specifies how far (in degrees) from the octant angle the arc begins.
- *endOffset* specifies how far from an octant angle the arc ends.
- *highRadius* specifies a radius larger than 255 units; when the arc has a radius of 255 units or smaller, then this parameter is 0. The *highRadius* is multiplied by 256, then added to the *radius* value to find the radius of the arc.
- *radius* is a value between 1 and 255.
- Negative sign changes the direction of the arc to clockwise; leave it out for counterclockwise direction.
- **0** specifies the following characters are hexadecimal.
- *startingOctant* specifies where the arc starts; the value ranges between 0 and 7.
- *octantSpan* specifies how hard the arc travels, again a number between 0 through 7.

TIP Here is how Autodesk suggests determining the value of *startOffset* and *endOffset*:

1. Determine the offsets by calculating the difference in degrees between the starting octant's boundary (which is always a multiple of 45 degrees) and the start of the arc.
2. Multiply the difference by 256.
3. Divide the result by 45.

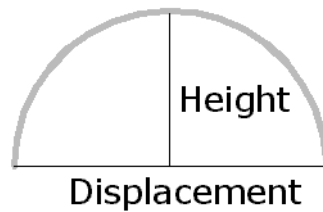
Bulge Arc - 12/00C

Code 12 draws a single-segment arc by applying a bulge factor to the displacement vector.

```
0C, xDisplacement, yDisplacement, bulge
```

- *xDisplacement* and *yDisplacement* specify the starting x,y-coordinates of the arc.
- *bulge* specifies the curvature of the arc. All three values range from -127 to 127.

This is how Autodesk says the bulge is calculated: “If the line segment specified by the displacement has length D, and the perpendicular distance from the midpoint of that segment has height H, the magnitude of the bulge is $((2 * H / D) * 127)$.”



A semicircle (180 degrees) would have a bulge value of 127 (drawn counterclockwise) or -127 (drawn clockwise), while line would have a value of 0. For an arc of greater than 180 degrees, use two arcs in a row.

Polyarc - 13/00D

Code 13 draws a *polyarc*, an arc with two or more parts. It is terminated by (0,0).

```
13, (0,2,127), (0,2,-127), (0,0)
```

TIP To draw a straight line between two arcs, it is more efficient to use a zero-bulge arc, than to switch between arcs and lines.

Flag Vertical Text Flag - 14/00E

Code 14 is for fonts only, and only fonts that are designed to be placed horizontally and vertically. When the orientation is vertical, the code following is processed; if horizontal, the code is skipped.

Notes

A series of horizontal dotted lines for writing notes, spanning the width of the page.

Using Script Files

In this chapter, we look at progeCAD's clearest customization possibility: the script. You also learn about progeCAD's script recording feature.

What are Scripts?

Scripts mimic what you type at the keyboard and click with the mouse in the drawing area. Anything you type in progeCAD that shows up at the ':' command prompt can be put in a script file. That includes progeCAD commands, their option abbreviations, your responses, and — significantly — LISP code. Other mouse actions, including selecting dialog box and toolbar buttons, *cannot* be included in a script file.

In This Chapter

- What are scripts?
- Drawbacks to scripts.
- Script commands and modifiers.
- Special characters.
- Recording scripts.

The purpose of the script is to reduce keystrokes by placing the keystrokes and screen coordinates in files. It was a predecessor to macros. For instance, a script file that draws a line and a circle looks like this:

```
line 1,1 2,2
circle 2,2 1,1
```

Script files have an extension of `.scr`. Script files are stored in plain ASCII. For that reason, I don't use a word processor, such as WordPad, OpenOffice, or Atlantis. Instead, I use Notepad to write scripts. Sometimes, when I feel like a DOS *power user*, I even write scripts at the DOS prompt:

```
C:\> copy con filename.scr
;This is the script file
line 1,1 2,2
circle 2,2 1,1
```

When I'm done, I press **F6** or **CTRL+Z** to tell DOS that I've finish editing, and to close the file.

Drawbacks to Scripts

A limitation to scripts is that only one script file can be loaded into progeCAD at a time. A script file can, however, call another script file. Or, you can use some other customization facility to load script files with a single mouse click, such as toolboxes, menu macros, and LISP routines.

Another limitation is that scripts stall when they encounter invalid command syntax. I sometimes have to go through the code-debug cycle a few times to get the script correct. It is useful to have an CAD reference text on hand that lists all command names and their options.

progeCAD includes a script creation command, **RecScript** (short for "record script").

Strictly Command-Line Oriented

There are two more limitations that are significant in this age of GUIs (graphical user interfaces): scripts cannot control mouse movements nor dialog boxes. For these reasons, nearly all commands that display a dialog box have a command line equivalent in progeCAD:

- Some commands have different names. For example, to control layers, there are the **Layer** (dialog box) and **-Layer** (command-line) commands. If the script needs to create or change a layer, use the **-Layer** command — or better yet — the **CLayer** system variable, as follows:
- Some commands need system variable **FileDia** turned off. This forces commands that display the **Open File** and **Save File** dialog boxes — such as **Open**, **Script**, and **VSlide** — to prompt for file names at the command line. Thus, script files should include the following lines to turn off file dialog boxes:

```
; Change layer:
clayer layername
```

```
; Turn off dialog boxes:
filedia 0
; Load slide file:
vslide filename
```

- When **FileDia** is turned off, use the ~ (tilde) as a filename prefix to force the display of the dialog box. For example:

```
: script
Script to run: ~ (progeCAD displays Run Script dialog box.)
```

Script Commands and Modifiers

There are a grand total of four commands that relate specifically to scripts. In fact, these commands are of absolutely no use for any other purpose. In rough order of importance, these are:

Script

The **Script** command performs double-duty: (1) it loads a script file; and (2) immediately begins running it. Use it like this:

```
: script
Script to run: filename
```

Remember to turn off (set to 0) the **FileDia** system variable, so that the prompts appear at the command line, instead of the dialog box.

RScript

Short for “repeat script,” this command reruns whatever script is currently loaded in progeCAD. A great way to create infinite loops. There are no options:

```
: rscript
```

Resume

This command resumes a paused script file. Pause a script file by pressing the **Backspace** key. Again, no options:

```
: resume
```

Delay

To create a pause in a script file without human intervention, use the **Delay** command along with a number. The number specifies the pause in milliseconds, where 1,000 milliseconds equal one second. The minimum delay is 1 millisecond; the maximum is 32767 milliseconds, which is just under 33 seconds.

While you could use **Delay** at the ‘:’ prompt, that makes little sense; instead, **Delay** is used in a script file to wait while a slide file is displayed or to slow down the script file enough for humans to watch the process, like this:

```
; Pause script for ten seconds:
delay 10000
```

Special Characters

In addition to these four script-specific commands, there are some special characters and keys.

Enter - (space)

The most important special characters are invisible: both the space and the carriage return (or end-of-line) represent you pressing the **SPACEBAR** and **ENTER** key. In fact, both are interchangeable. But the tricky part is that they are invisible.

Sometime, I'll write a script that requires a bunch of blank space because the command requires that I press the **ENTER** key several times in a row. **AttEdit** is an excellent example:

```
; Edit the attributes one at a time:  
attedit 1,2
```

How many spaces are there between **attedit** and the coordinates **1,2**? I'll wait while you count them...

For this reason, it is better to place one script item per line, like this:

```
; Edit the attributes one at a time:  
attedit
```

```
1,2
```

Now it's easier to count those four spaces, since there is one per blank line.

Comment - ;

You probably have already noticed that the semicolon lets you insert comments in a script file. progeCAD ignores anything following the semicolon.

Transparent - '

Scripts can be run *transparently* during a command. Simply prefix the **Script** command to run a script while another command is active, like this:

```
: line  
Start of line: 'script  
Script to run: filename
```

All four of progeCAD's script-specific commands are transparent, even **'Delay**. That lets you create a delay during the operation of a command — as if I needed an excuse to run progeCAD slowly!

Pause - Backspace

...is the key I mentioned earlier for pausing a script file.

Stop - ESC

...stops a script file dead in its tracks; use the **RScript** command to start it up again from the beginning

Recording Scripts

progeCAD has the ability to record scripts. That can make it easy to write scripts, although you may need to edit out typing errors. The **RecScript** command records all your keystrokes and mouse picks in the drawing area (i.e., x,y-coordinate picks.)

The script recorder does not, however, record your use of the mouse with menus, toolbars, and dialog boxes. progeCAD's documentation warns, "Using these while recording a script causes unpredictable results." So, command-line activity only!

With that in mind, here are the steps to recording a script:

1. Turn off dialog boxes, as follows:

```
: filedia  
New current value for FILEDIA (Off or On) <On>: off
```

2. From the menu bar, select Tools | Record Script. progeCAD prompts you to provide a name for the script:

```
: RECSRIPT  
Script to record: (Enter a filename.)
```

Remember to include the *.scr* extension, otherwise progeCAD leaves it out. For example, enter *tailoring.scr*. The name you provide becomes the filename for the saved *.scr* file.

3. progeCAD asks if you want to append the script to an existing *.scr* file — curious that it asks, even if this is a new filename you are proposing:

```
Append to script? <N>: (Press ENTER.)
```

4. Enter commands and options. When you pick points in the drawing with the mouse, progeCAD records the pick points as coordinates, like the following:

Script Command	Comments
c 5.0276,6.2169	Enter Circle command, pick center point...
7.1124,4.1321	...and pick radius point.
l 1.2435,6.7950	Enter Line command, pick first point...
8.4613,6.7074	...pick second point...
2.8202,2.2576	... and pick third point.

5. When done, go to the **Tools** menu bar, and then select **Stop Recording**.
6. Turn on **FileDia** to make dialog boxes visible again.
7. To run the recorded script file, use the **Script** command.

Programming LISP

While toolbar and menu macros are easy to write and edit, they limit your ability to control progeCAD. In this chapter, we look at the most powerful method available to “non-programmers” for customizing progeCAD — the LISP programming language — at the cost of being somewhat more difficult to create than macros or scripts.

While LISP is meant for writing programs, it can be used like macros. This chapter shows you both approaches.

In This Chapter

- The history of LISP in IntelliCAD.
- Compatibility between LISP and AutoLISP.
- The LISP programming language.
- Simple LISP: adding two numbers.
- LISP in commands.
- LISP function overview.
- External command functions.
- Accessing system variables.
- Advanced LISP functions.
- Writing a simple LISP program.
- Saving data to files.
- Tips in using LISP.

The History of LISP in CAD

LISP is one of the earliest programming languages, developed in the late 1950s to assist artificial intelligence research. Its name is short for “list processing,” and it was designed to handle lists of words, numbers, and symbols.

LISP first appeared in CAD when, back in 1985, Autodesk added an undocumented feature to AutoCAD v2.15 called “Variables and Expressions.” Programmers at Autodesk had taken XLISP, a public domain dialect written by David Betz, and adapted it for AutoCAD. The initial release of Variables and Expressions was weak, because it lacked *conditional statements* — needed by programming languages to make decisions.

With the release AutoCAD v2.5, however, Autodesk got serious. They added the missing programming statements; they added the powerful GETxxx, SSxxx, and EntMod routines that provide direct access to entities in the drawing database; and they renamed the programming language “AutoLISP.” This allowed third-party developers to write routines that manipulated the entire drawing, and non-programmers to write simple routines that automated everyday drafting activities.

When SoftDesk developed IntelliCAD, they included a programming language very similar to AutoLISP, calling it simply “LISP.” (I think it would have been better to call it IntelliLISP to prevent confusion with the real LISP programming language. Better yet, they could have given it the trendy moniker of iLISP.)

Since Autodesk’s original idea, other CAD packages also included dialects of LISP, including CadKey, TurboCAD, and FelixCAD. Later, Autodesk added a compiled version of AutoLISP to AutoCAD, called Visual LISP.

In addition to LISP, you can also write programs for progeCAD using SDS (solution development system) and VBA (visual basic for applications).

Compatibility between LISP and AutoLISP

LISP in progeCAD is, for the most part, compatible with AutoCAD’s AutoLISP. If you know AutoLISP, you can program immediately in LISP, including controlling dialog boxes. LISP has, however, some differences you should be aware of.

Additional LISP Functions

LISP contains several additional functions not found in AutoLISP. These include the following:

LISP Function	Comment
log10	Returns log base 10 of the number.
lpad	Pads the text string with spaces to the left.
rpad	Pads the text string with spaces to the right.
tan	Returns the tangent of the angle.
trim	Trims spaces from the string.

Different LISP Functions

LISP has several functions that operate differently from AutoLISP, by providing additional support. These include the following:

LISP Function	Comment
osnap	Supports PLA (planview) entity snap for snapping to 2D intersections.
ssget and ssadd	Supports additional selection modes: <ul style="list-style-type: none">CC - Crossing CircleO - OutsideOC - Outside CircleOP - Outside PolygonPO - PPoint

Missing AutoLISP Functions

LISP lacks some functions found in AutoLISP. These include: `acdmenableupdate`, `acet-attdsync`, `acet-layerp-mode`, `acet-layerp-mark`, `acet-laytrans`, `acet-ms-to-ps`, `acet-ps-to-ms`, `defun-q`, `defun-q-list-ref`, `defun-q-list-set`, `entmakex`, `initdia`, `namedobjdict`, `ssnamex`, and `tablet`.

Also missing are:

- All ARX-related functions that run ObjectARX applications
- All **dict**-related functions.
- All **vl**-related functions for Visual LISP.
- All SQL-related functions, which link between objects in the AutoCAD drawing with records in an external database file. In AutoCAD, these functions start with “`ase_`”, as in `ase_lsunite` and `ase_docmp`.

The LISP Programming Language

LISP is capable of many masks, from adding together two numbers — during the middle of a command — to drawing parametrically a staircase in 3D, to generating a new user interface for progeCAD, to manipulating data in the drawing database.

The most important aspect of LISP, in my opinion, is that it lets you toss off a few lines of code to help automate your work. In this chapter’s tutorials, I show you how to write simple LISP code that makes your progeCAD drafting day easier.

In contrast, IntelliCAD’s most powerful programming facility — known as SDS (solutions development system) — is merely an interface: you have to buy additional the programming tools (read: \$\$\$) and have an in-depth knowledge of advanced programming methodology. The primary advantage to using SDS is speed: these programs run compute-intensive code as much as 100 times faster than LISP.

Simple LISP: Adding Two Numbers

With that bit of background, let’s dive right into using LISP. Let’s start with something easy, something everyone knows about, adding together two numbers, like 9 plus 7.

1. Start progeCAD, any version.
2. When the 'Command:' command prompt appears, type the boldface text, shown below, on the keyboard:

```
Command: (+ 9 7)      (Press Enter.)
16
Command:
```

progeCAD instantly replies with the answer, 16. (In this chapter, I show the function we're talking about in **cyan**.)

Getting to this answer through (+ 9 7) may, however, seem convoluted to you. That's because LISP uses *prefix notation*:

The **operator**, +, appears before the **operands**, 9 and 7.

Think of it in terms of "add 9 and 7." This is similar to how progeCAD itself works: type in the command name first (such as Circle), and then enter the coordinates of the circle.

3. Notice the parentheses that surround the LISP statement. Every opening parenthesis, (, requires a closing parenthesis,). I can tell you right now that balancing parentheses is the most frustrating aspect to LISP. Here's what happens when you leave out the closing parentheses:

```
Command: (+ 9 7      (Press Enter.)
Missing: 1) >
```

progeCAD displays the "Missing: 1)" prompt to tell you that one closing parenthesis is missing. If two closing parentheses were missing, the prompt would read "Missing: 2)".

4. Type the missing) and progeCAD is satisfied:

```
Missing: 1) >) (Press ENTER.)
16
Command:
```

5. The parentheses serve a second purpose: they alert progeCAD that you are using LISP. If you were to enter the same LISP function '+ 7 9' without parentheses, progeCAD would react unfavorably to each character typed, interpreting each space as the end of a command name:

```
Command: + (Press the spacebar.)
Unable to recognize command. Please try again.
Command: 9 (Press the spacebar.)
Unable to recognize command. Please try again.
Command: 7 (Press the spacebar.)
Unable to recognize command. Please try again.
Command:
```

6. As you might suspect, LISP provides all the basic arithmetic functions: addition, subtraction, multiplication, and division. Try each of the functions, subtraction first:

```
Command: (- 9 7)
2
Command:
```

7. Multiplication is done using the familiar * (asterisk) symbol, as follows:

```
Command: (* 9 7)
63
Command:
```


8. Finally, division is performed with the / (slash) symbol:

```
Command: (/ 9 7)
1
Command:
```

Oops, that's not correct! Dividing 9 by 7 is 1.28571, not 1. What happened? Up until now, you have been working with *integer* numbers (also known as *whole* numbers). For that reason, LISP has been returning the results as integer numbers, although this was not apparent until you performed the division.

To work with real numbers, add a decimal suffix, which can be as simple as .0 — this converts integers to real numbers, and forces LISP to perform real-number division, as follows:

```
Command: (/ 9.0 7)
1.28571
Command:
```

And LISP returns the answer correct to five decimal places.

9. Finally, let's see how LISP lets you *nest* calculations. "Nest" means to perform more than one calculation at a time.

```
Command: (+ (- (* (/ 9.0 7.0) 4) 3) 2)
4.14286
Command:
```

Note how the parentheses aid in separating the nesting of the terms.

LISP in Commands

Okay, so we've learned how progeCAD works as a \$695 four-function calculator. This overpriced calculator pays its way when you employ LISP to perform calculations within commands. For example, you may need to draw a linear array of seven circles to fit in a 9" space.

1. Start the **Circle** command, as follows:

```
Command: circle
2Point/3Point/RadTanTan/Arc/Multiple/<Center of circle>: (Pick a point.)
```

2. Instead of typing the value for the diameter, enter the LISP equation, as follows:

```
Diameter/<Radius>: (/ 9.0 7)
Diameter/<Radius>: 1.28571
```

progeCAD draws a circle with a diameter of 1.28571 inches. You can use an appropriate LISP function anytime progeCAD expects user input.

3. Now go on to the **-Array** command, and draw the other six circles, as follows:

```
: -array
Select entities to array: L
Entities in set: 1 Select entities to array: (Press Enter.)
Type of array: Polar/<Rectangular>: r
Number of rows in the array <1>: (Press Enter.)
Number of columns <1>: 7
Horizontal distance between columns: (/ 0.9 7)
Horizontal distance between columns: 0.128571
```

Once again, you use LISP to specify the array spacing, which happens to equal the circle diameter.

Remembering the Result: `setq`

In the above example, you used the `(/ 9.0 7)` equation twice: once in the Circle command and again in `-Array`. Just as the **M**-key on a calculator lets it remember the result of your calculation, LISP can be made to remember the results of *all* your calculations.

To do this, employ the most common LISP function, known as **setq**. This curiously named function is short for *SET eQual to*.

1. To save the result of a calculation, use the **setq** function together with a *variable*, as follows:

```
Command: (setq x (/ 9.0 7))
1.28571
Command:
```

Here, **x** remembers the result of the `(/ 9.0 7.0)` calculation. Notice the extra set of parentheses.

From algebra class, you probably recall equations like ' $x = 7 + 9$ ' and ' $x = 7 / 9$ '. The *x* is known as a *variable* because it can have any value.

2. To prove to yourself that *x* contains the value of 1.28571, use IntelliCAD's **!** (exclamation) prefix, as follows:

```
Command: !x
1.28571
Command:
```

The **!** prefix (sometimes called “the bang”) is useful for reminding yourself of the value contained by a variable, in case you've forgotten, or are wondering what happened during the calculation.

LISP isn't limited to just one variable. You can make up any combination of characters to create variable names, such as **pt1**, **diameter**, and **yvalue**. The only limitation is that you cannot use LISP function names, such as **setq**, **T**, and **getint**. In fact, it is good to create variable names that reflect the content, such as the circle diameter calculated above. But you also want to balance a descriptive name, such as **diameter**, with minimized typing, such as **x**. A good compromise for our example is **dia**.

3. You make one variable equal another, as follows:

```
Command: (setq dia x)
1.28571
Command: !dia
1.28571
Command:
```

4. Redo the Circle and `-Array` commands, this time using variable **dia**, as follows:

```
Command: circle
2Point/3Point/RadTanTan/Arc/Multiple/<Center of circle>: (Pick a point.)
Diameter/<Radius>: !dia
Diameter/<Radius>: 1.28571
Command: -array
```

```
Select entities to array: L
Entities in set: 1 Select entities to array: (Press Enter.)
Type of array: Polar/<Rectangular>: r
Number of rows in the array <1>: (Press Enter.)
Number of columns <1>: 7
Horizontal distance between columns: dia
Horizontal distance between columns: 0.128571
```

progeCAD draws precisely the same seven circles, using the value 1.28571 stored in **dia**.

LISP Function Overview

LISP is so powerful that it can manipulate almost any aspect of the progeCAD drawing. In the following tutorial, you get a taste of the many different kinds of functions LISP offers you for manipulating numbers and words. As we start on our whirlwind tour of several groups of functions, start progeCAD, and then type the examples in the Prompt History window by pressing **F2** at the 'Command:' command prompt.

Math Functions

In addition to the four basic arithmetic functions you already learned, LISP has many of the mathematical functions you might expect in a programming language. The list includes trigonometric, logarithmic, logical, and bit manipulation functions; one type of function missing is matrix manipulation.

For example, the **min** function returns the smallest (minimum) of a list of numbers:

```
Command: (min 7 3 5 11)
3
```

To remember the result of this function, add **setq** with variable **minnbr**, as follows:

```
Command: (setq minnbr (min 7 3 5 11))
3
```

Now each time you want to refer to the minimum value of that series of numbers, you can refer to variable **minnbr**. Here's an example of a trig function, sine:

```
Command: (sin minnbr)
0.14112
```

Returns the sine of the angle of 3 radians.

TIPS You must provide the angle in **radians**, not degrees. This is many times an inconvenience, because often you work with degrees, but must convert them to radians.

Fortunately, LISP can do this for you, as long as you code it correctly. Recall that there are **2*pi** (approximately 6.282) radians in 360 degrees. For example, to get the sine of 45 degrees, you have to indulge in some fancy footwork:

```
Command: (sin (* (/ 45 180.0) pi))
0.707107
```

Here I divided the degrees (45) by 180, then multiplied by **pi**. Either the 45 or the 180 needs a decimal (.0) to force division by real numbers, rather than by integers.

By the way, **pi** is the only constant predefined in LISP, and is equal to 3.1415926. That means you just type **pi**, instead of 3.1415926 each time you need the value of pi in a function. To see this for yourself, use the exclamation mark at the command prompt:

```
Command: !pi
3.14159
```

LISP displays the result to six decimal places, even though it performs calculations to 32-bit accuracy.

Geometric Functions

Since CAD deals with geometry, LISP has a number of functions for dealing with geometry.

Distance Between Two Points

The LISP **distance** function is similar to progeCAD's **Dist** command: it returns the 3D distance between two points. To see how it works, first assign x,y-coordinates to a pair of points, **p1** and **p2**, as follows:

```
Command: (setq p1 '(1.3 5.7))
(1.3 5.7)
Command: (setq p2 '(7.5 3.1 11))
(7.5 3.1 11)
Command: (distance p1 p2)
6.72309
```

You may have missed that single quote mark in front of the list of x,y-coordinates, as in: '(1.3 5.7). That tells LISP you are creating a *pair* (or *triple* in the case of x,y,z) of coordinates, and that it should not evaluate the numbers. Technically, the ' mark creates a *list* of numbers.

To separate the coordinates use spaces, not commas. Note that when you leave out the z-coordinate, LISP assumes it equals 0.0000.

The Angle from 0 Degrees

Other geometric functions of interest include finding the angle from 0 degrees (usually pointing east) to the line defined by **p1** and **p2**:

```
Command: (angle p1 p2)
5.88611
```

The result is returned in radians: 5.88611.

The Intersection of Two Lines

The intersection of two lines is determined by the **inters** function:

```
Command: (inters pt1 pt2 pt3 pt4)
```

Entity Snaps

In the following function, you are finding the midpoint of the line that starts at **p1**. You apply the **osnap** function and specify the type of osnap; LISP returns the x,y,z-coordinates of the entity snap point. The entity must actually exist.

```
Command: line
From point: lp1
To point: lp2
To point: (Press ENTER.)
Command: (osnap p1 "mid")
(4.4 4.4 5.5)
```

Here "**mid**" refers to the midpoint entity snap mode.

Other geometric functions include **textbox** (for finding the rectangular outline of a line of text) and **Polar**, which returns a 3D point of a specified distance and angle.

Conditional Functions

You could say that *conditional* functions are most important, because they define the existence of a programming language. It is conditionals that allow a computer program to “think” and make decisions. Conditional functions check if one value is less than, equal to, or greater than another value. They check if something is true; or they repeat an action until something is false.

If you’re not sure if it’s a programming language or merely a macro language, check for conditionals. Toolbar macros, for example, have no conditionals; they are not a programming language.

Here is an example of how conditional functions operate: *if* the floor-to-ceiling distance *is greater than eight feet, then* draw 14 steps; *else*, draw 13 steps. Notice that there are two parts to the statement: the *if* part is the true part; the *else* part is the false part. Do something if it is true; otherwise, so something else if it is false.

Similar wording is used in LISP’s condition functions. Enter the following at the ‘:’ prompt:

```
Command: (if (> height 96) (setq steps 14) (setq steps 13))
13
```

Let’s break down this code to see how the **if** function compares with our statement:

(if	if
(>	greater than
height	floor-to-ceiling distance is
96)	8 feet;
(setq steps 14)	Then
(setq steps 13)	use 14 steps.
)	Else
	use 13 steps.

Other Conditionals

The **if** function is limited to evaluating just one conditional. The **cond** functions evaluate many conditions. The **repeat** function executes a specific number of times, while the **while** function executes code for as long as it is true.

String and Conversion Functions

You can manipulate *strings* (text consisting of one or more characters) in LISP, but to a lesser extent than numbers. For example, you can find the length of a string as follows:

```
Command: (strlen "progeCAD World")
17
```

The **strlen** (short for *STRing LENgth*) function tells you that “progeCAD World” has 17 characters in it, counting the space. Notice how “progeCAD World” is surrounded by quotation marks. That tells LISP you are working with a string, not a variable.

If you were to type **(strlen progeCAD World)**, LISP tries to find the length of the strings held by variables `progeCAD` and `World`. For example:

```
Command: (setq progeCAD "A software package")
"A software package"
Command: (setq world "the planet earth")
"the planet earth"
```

Command: (strlen progeCAD world)

34

Joining Strings of Text

Other string functions change all characters to upper or lower case (**strcase**), returns part of a string (**substr**), searches and replaces text in a string (**subst**), and join two strings together (**strcat**), as follows:

Command: (strcat progeCAD " used all over " world)

"A software package used all over the planet earth"

That's how you create reports, such as "13 steps drawn", by mixing variables and text.

Converting Between Text and Numbers

Related to string functions are the conversion functions, because some of them convert to and from strings. For example, earlier I showed how to convert degrees to radians. That's fine for decimal degrees, like 45.3711 degrees. But how do you convert 45 degrees, 37 minutes and 11 seconds, which progeCAD represents as 45d37'11"? That's where a conversion function like **angtof** (short for *ANGLE TO Floating-point*) comes in. It converts an angle string to real-number radians:

Command: (angtof "45d37'11\" 1)

0.796214

Here we've supplied **angtof** with the angle in degrees-minutes-seconds format. However, LISP isn't smart enough to know, so we tell it by means of the *mode* number, 1 in this case. This (and some other functions) use the following as mode codes:

Mode	Meaning	Example
0	Decimal degrees	45.3711
1	Degrees-minutes-seconds	45d 37' 11"
2	Grad	100.1234
3	Radian	0.3964
4	Surveyor units	N 45d37'11" E

Notice the similarity between the mode numbers and the values of system variable **AUnits**. The coincidence is not accident. When you don't know ahead of time the current setting of units, you make use of this fact by specifying the mode number as a variable, as follows:

Command: (angtof "45d37'11\" (getvar "aunits"))

0.796214

Here we use **getvar** (short for *GET VARIABLE*), the LISP function that gets the value of a system variable. We used **getvar** to get **aunits**, which holds the state of angular display as set by the **Units** command.

Notice how the seconds indicator (") is handled: \". That's so it doesn't get confused with the closing quote mark (") that indicates the end of the string.

Other Conversion Functions

Other conversion functions convert one unit of measurement into another (via the **cvunit** function and the *icad.unt* file), an integer number into a string (**itos**), a character into its ASCII value (**ascii**: for example, letter A into ASCII value 65), and translates (moves) a point from one coordinate system to another (**trans**).

External Command Functions

“Powerful” often equates to “complicated,” yet one of LISP’s most powerful functions is its simplest to understand: the **command** function. As its name suggests, **command** executes progeCAD commands from within LISP.

Think about it: this means that it is trivial to get LISP to draw a circle, place text, zoom a viewport, whatever. Anything you *type* at the ‘Command:’ command prompt is available with the **command** function. Let’s see how **command** works by drawing a circle. First, though, let’s recall how the **Circle** command operates:

```
Command: circle
2Point/3Point/RadTanTan/Arc/Multiple/<Center of circle>: 2,2
Diameter/<Radius>: D
Diameter of circle: 1.5
```

Switching to the **command** function, you mimic what you type at the ‘Command:’ prompt, as follows. (This is where Chapter 8’s practice in creating script files is handy.)

```
Command: (command "circle" "2,2" "D" "1.5")
```

Notice how all typed text is in quotation marks. After you enter that line of code, progeCAD responds by drawing the circle:

```
Command: circle
2Point/3Point/RadTanTan/Arc/Multiple/<Center of circle>: 2,2
Diameter/<Radius> <1.2857>: D
Diameter of circle <2.5714>: 1.5
```

Let’s look at one of the more complex commands to use with the **command** function, Text. When we use the Text command, progeCAD presents these prompts:

```
Command: text
Text: Style/Align/Fit/Center/Middle/Right/Justify/<Start point>: 5,10
Height of text <0.2000>: 1.5
Rotation angle of text <0>: (Press ENTER.)
Text: Tailoring progeCAD
```

Converted to LISP-ese, this becomes:

```
Command: (command "text" "5,10" "1.5" "" "Tailoring progeCAD")
```

And progeCAD responds with:

```
Command: text
Text: Style/Align/Fit/Center/Middle/Right/Justify/<Start point>: 5,10
Height of text <1.5000>: 1.5
Rotation angle of text <0>:
Text: Tailoring progeCAD
```

...and then draws the text.

For the ‘Rotation angle:’ prompt, we had simply pressed the Enter key. Notice how that is dealt with in the LISP function: "" — a pair of empty quotation marks.

You use the same "" to end commands that automatically repeat themselves, such as the **Line** command:

```
Command: (command "line" "1,2" "3,4" "")
```

When you don’t include that final "", then you leave progeCAD hanging with a ‘End point:’ prompt and your LISP routine fails.

By now it should be clear to you that you have to really know the prompt sequence of IntelliCAD’s more than 300 commands to work effectively with the **command** function. The easiest way to get a handle on those is to purchase one of the “quick reference” books on the market, which

list commands in alphabetical order, along with the complete prompt sequence. And, as we see in a minute, check that the quick reference book has a listing of all system variables, their default value, and the range of permissible values.

Command Function Limitation

But the **command** function has a failing. Earlier, I said, “Anything you *type* at the ‘Command:’ command prompt is available with the **command** function.” I now place emphasis on the word “type.” The **command** function breaks down completely when it comes to dialog boxes. That’s right: any command that uses a dialog box won’t work with the command function — nor, for that matter, with the macros we looked at in previous chapters. It is for this reason that progeCAD includes command-line versions of almost every (but not all) command.

Accessing System Variables

While you can use the **command** function to access system variables, LISP has a pair of more direct functions: **getvar** and **setvar**. **Getvar** gets the value of a system variable, while **setvar** changes (sets) the value.

For example, system variable **SplFrame** determines whether the frame of a spline polyline is displayed; by default, the value of **SplFrame** is 0: the frame is not displayed, as confirmed by **getvar**:

```
Command: (getvar "splframe")  
0
```

To display the frame, change the value of **SplFrame** to 1 with **setvar** as follows:

```
Command: (setvar "splframe" 1)  
1
```

We have, however, made a crass assumption: that the initial value of **SplFrame** is 0. Zero is the default value, but not necessarily the value at the time that you run the LISP routine. How do we know what the value of **SplFrame** is before we change it? We’ll answer that question later in this chapter. Stay tuned.

GetXXX Functions

It’s one thing to execute a command that draws a new entity, such as the circle and text we drew above with the **command** function. It is trickier working with entities that already exist, such as moving that circle or editing the text. That’s where the a group of functions known collectively as **Getxxx** come into play. These functions get data from the screen. Some of the more useful ones include:

getpoint	Returns the x,y,z-coordinate of a picked point.
getangle	Returns the angle in radians.
getstring	Returns the text typed by the user.
getreal	Returns the value of a real number typed by the user.

Here’s how to use some of these with the **Text** command. Let’s redo the code with **getstring** so that LISP prompts us for everything first, then executes the **Text** command. Here is the first line of code, which prompts the user to input some text:

```
Command: (setq TxtStr (getstring T "What do you want to write? "))  
What do you want to write?
```


Notice that extra "T"; that's a workaround that lets **getstring** accept a string of text with spaces. When you leave out the **T**, then **getstring** accepts text up to the first space only. If you were to enter "Tailoring progeCAD", you would end up with just "Tailoring" and no "progeCAD."

Also in the line of code above, the **setq** function stores the phrase, such as "Tailoring progeCAD," in the variable **TxtStr**.

In the next line of code, we use the **getreal** function to ask for the height of text, which is a real number (decimal) entered by the user.

```
Command: (setq TxtHt (getreal "How big do you want the letters? "))
How big do you want the letters? 2
2.0
```

Notice how **getreal** converts the 2 (an integer) to a real number, 2.0. The value is stored in variable **TxtHt**.

Next, we use the **getangle** function to ask for the rotation angle of the text:

```
Command: (setq TxtAng (getangle "Tilt the text by how much? "))
Tilt the text by how much? 30
0.523599
```

Notice how **getangle** converts the 30 (a decimal degree) into radians, 0.523599. The value is stored in variable **TxtAng**.

Next, we use the **getpoint** function to ask the user for the insertion point of the text:

```
Command: (setq TxtIns (getpoint "Where do you want the text to start? "))
Where do you want the text to start? (Pick a point.)
(27.8068 4.9825 0.0)
```

Notice how **getpoint** returns the x, y, and z values of the coordinate, even though z is zero. The user can pick a point on the screen, or enter a coordinate pair (x,y) or triple (x,y,z).

Finally, we execute the **Text** command with the four variables:

```
Command: (command "text" TxtIns TxtHt TxtAng TxtStr)
text Justify/Style:
Height <1.5000>: 2.0000000000000000
Rotation angle <0>: 0.523598775598299
Text: Tailoring progeCAD
: nil
```

There! We've just customized the **Text** command to our liking. Not only did we change the prompts that the user sees, but we used LISP to change the order of the prompts.

Selection Set Functions

To work with more than one entity at a time, LISP has a group of functions for creating selection sets. These all begin with "SS", as in:

SsAdd	Adds entities to selection sets.
SsDel	Deletes entities from selection sets.
SsGetFirst	Reports the number of selected entities.
SsLength	Reports the number of entities in the selection set.
SsMemb	Checks if entities are part of a selection set.
SsName	Identifies the nth entity in a selection set.
SsSetFirst	Highlights objects in a selection set.

IntelliCAD's **Select** command can deal only with one selection set at a time; in contrast, the LISP SSxxx commands can work with up to 128 selection sets.

Entity Manipulation Functions

The really powerful LISP functions are the ones that go right in and manipulate the drawing database. Unlike the **command** function, which is powerful but simple, the entity manipulation functions are powerful and complicated. Here's a summary of what some of these are:

EntMake	Creates new entities.
EntGet	Gets the data that describes entities in drawings.
EntMod	Changes entities.
EntDel	Erases entities from the database.
TblObjName	Gets the names of entities in symbol tables.

The “Ent” prefix is short for entity. The “symbol table” refers to the part of the drawing database that stores the names of layers, text styles, and other named entities in the drawing.

To create and manipulate entities, these LISP functions work with a variant on the DXF format, known as “dotted pairs.” For example, to work with a layer named **RightOfWay**, you employ the following format:

`"2 . RightOfWay"`

The quotation marks indicate the start and end of the data, while the dot in the middle separates the two values: The **2** is the DXF code for layer names, and **RightOfWay** is the name of the layer. You can see that to work with these entity manipulation functions, you need a good grasp of the DXF format.

Advanced LISP Functions

There is a whole host of LISP functions that you may never use in your progeCAD programming career. For example, there are LISP functions for controlling the memory, such as **as gc** (garbage collection) and **mem** (memory status). Another set of LISP functions are strictly for loading and displaying dialog boxes, such as **load_dialog** and **new_dialog**.

Writing a Simple LISP Program

In this section, you learn the first steps in writing a LISP routine of your own.

Why Write a Program?

If you are like many CAD users, you are busy creating drawings, and you have no time to learn how to write software programs. No doubt, you may be wondering, “Why bother learning a programming language?” In some ways, it’s like being back again in school. Sitting in the classroom sometimes seems like a waste of time.

But the things you learn now make life easier later. Learning some LISP programming now means you’ll feel really good whipping off a few lines of code to let LISP perform tedious tasks for you. The nice thing about LISP is that you can program it on the fly. And you can use it for really simple but tedious tasks.

Here’s the example we’ll use for this tutorial:

The Id Command

progeCAD has the **Id** command. When you pick a point on the screen, **Id** reports the 3D x,y,z-coordinates of the point. Problem is, **Id** reports the value in the command prompt area, like this:

```
Command: id  
Select a point to identify coordinates: (Pick a point.)  
X = 8.9227 Y = 6.5907 Z = 0.0000
```

Wouldn’t it be great if you could change **Id** so that it places the coordinates in the drawing, next to the pick point? That would let you label x,y-coordinates and z-elevations over a site plan. With LISP, you can.

The Plan of Attack

Before you write any LISP code, you need to figure out how you’re going to get those x,y,z-coordinates off the command prompt area, and into the drawing. Recognize that there are two parts to solving the problem:

Part 1. Obtain the coordinates from the drawing, probably by picking a point.

Part 2. Place the coordinates as text in the drawing.

Obtaining the Coordinates

LISP provides several ways to get the coordinates of a picked point. Browsing through the *LISP Programming Language Reference*, you learn you could:

- Use the **Id** command with the **command** function, as in (**command "ID"**).
- Use the **LastPoint** system variable with the **getvar** function, as in (**getvar "lastpoint"**).
- Use the **getpoint** function, as in (**getpoint "Pick a point: "**)

It would be a useful lesson to use each of the three, and then see what happens. By experimenting, you make mistakes, and then learn from the mistakes.

1. Start progeCAD, load a drawing, and switch to the Prompt History window with **F2**.

At the ':' prompt, enter:

```
Command: (command "ID")
```

Here you are executing an progeCAD command (**Id**) from within a LISP routine. The **command** function lets you use any progeCAD command in LISP. The progeCAD command is in quotation marks "**ID**" because the command is a *string* (programmer-talk for "text"). Just as before, progeCAD prompts you for the point.

2. In response to the LISP routine's prompt, pick a point:

```
Select a point to identify coordinates: (Pick a point.)
```

```
X = 8.9227 Y = 6.5907 Z = 0.0000
```

3. Unknown to you, progeCAD always stores the x,y,z-coordinates of the last-picked point in a system variable called **LastPoint**. So, you should copy the coordinates from **LastPoint** to a variable of your own making. You need to do this because the coordinates in **LastPoint** are overwritten with the next use of a command that makes use of a picked point.

Recall from in this chapter that the **setq** function stores values in variables. Make use of it now. At the ':' prompt, enter:

```
Command: (setq xyz (getvar "LastPoint"))
```

```
(8.9227 6.5907 0.0000)
```

Xyz is the name of the variable in which you store the x,y,z-coordinate.

Getvar is the name of the LISP function that retrieves the value stored in a system variable.

And "**LastPoint**" is the name of the system variable; it is surrounded by quotation marks because it is a system variable name (a string).

After entering the LISP function, progeCAD returns the value it stored in variable **xyz**, such as (8.9227 6.5907 0.0000) — your result will be different. Notice how the coordinates are surrounded by parenthesis. This is called a *list*, for which LISP is famous (indeed, LISP is short for "list processing"). Spaces separate the numbers, which are the x, y, and z-coordinates, respectively:

```
x 8.9227
y 6.5907
z 0.0000
```

progeCAD always stores the values in the order of x, y, and z. You will never find the z- coordinate first or the x-coordinate last.

So, we've now solved the first problem in one manner. We obtained the coordinates from the drawing, and then stored them in a variable. We did mention a third LISP function we could use, **getpoint**. Programmers prefer **getpoint** because it is more efficient than the **Id-LastPoint** combo we used above.

Type the following to see that it works exactly the same, the difference being that we provide the prompt text ("Point: "):

```
Command: (setq xyz (getpoint "Point: "))
Point: (Pick a point.)
(8.9227 6.5907 0.0000)
```

As before, we use the **setq** function to store the value of the coordinates in variable **xyz**. The **getpoint** function waits for you to pick a point on the screen. The "**Point:** " is called a prompt, which tells the user what the program is expecting the user to do. We could just as easily have written anything, like:

```
Command: (setq xyz (getpoint "Press the mouse button: "))
Press the mouse button: (Pick a point.)
(8.9227 6.5907 0.0000)
```

Or, we could have no prompt at all, as follows:

```
Command: (setq xyz (getpoint))
(Pick a point.)
(8.9227 6.5907 0.0000)
```

That's right. No prompt. Just a silent progeCAD waiting patiently for the right thing to happen ... and the user puzzled at why nothing is happening. A lack of communication, you might say. That's why prompts are important.

We've now seen a couple of approaches that solve the same problem in different ways. With the x,y,z-coordinates safely stored in a variable, let's tackle the second problem

Placing the Text

To place text in the drawing, we can use only the **command** function in conjunction with the **Text** command. I suppose the **MText** command might work, but you want to place one line of text, and the **Text** command is excellent for that. The **Text** command is, however, trickier than the **Id** command. It has a minimum of four prompts that your LISP routine must answer:

```
Command: text
Text: Style/Align/Fit/Center/Middle/Right/Justify/<Start point>:
Height of text <2>:
Rotation angle of text <0>:
Text:
```

- **Start point:** a pair of numbers, specifically an x,y-coordinate.
- **Height of text:** a number to makes the text legible.
- **Rotation angle of text:** a number, probably 0 degrees.
- **Text:** the string, in our case the x,y,z-coordinates.

Let's construct a LISP function for placing the x,y,z-coordinates as text:

```
(command "text" xyz 200 0 xyz)
```

(command is the **command** function.

"text" is the progeCAD **Text** command being executed.

xyz variable stores the starting point for the text.

200 is the height of the text. Change this number to something convenient for your drawings.

0 is the rotation angle of the text.

xyz means you're lucky: the Text command accepts numbers as text.

) and remember: one closing parenthesis for every opening parenthesis.

Try this out at the ':' prompt:

```
Command: (command "text" xyz 200 0 xyz)
Text: Style/Align/Fit/Center/Middle/Right/Justify/<Start point>:
Height of text: 200
Rotation angle of text: 0
Text: 2958.348773815669,5740.821183398367
Command:
```

progeCAD runs through the **Text** command, inserting the responses for its prompts, then placing the coordinates as text. We've solved the second part of the problem.

Putting It Together

Let's put together the two solutions to your problem:

```
(setq xyz (getpoint "Pick point: "))
(command "text" xyz 200 0 xyz)
```

There you have it: a full-fledged LISP program. Well, not quite. It's a pain to retype those two lines each time you want to label a point. In the next section, you find out how to save the code as a *.lsp* file on disk. You'll also dress up the code.

Adding to the Simple LISP Program

There you have it: a full-fledged LISP program. Well, not quite. What you have is the *algorithm* — the core of every computer program that performs the actual work. What is lacking is most of a *user interface* — the part that makes it easier for any user to employ the program.

All you have for a user interface is part of the first line that prompts, “Select point to identify coordinates:”. There’s a lot of user interface problems with this little program. How many can you think of? Here’s a list of problems I came up with:

- It’s a pain to retype those two lines each time you want to label a point — you need to give the program a name ...
- ... and you need to save it on disk so that you don’t need to retype the code with each new progeCAD session...
- ... and, if you use this LISP program a lot, then you should have a way of having it load automatically.
- The x,y,z-coordinates are printed to eight decimal places; for most users, that’s w-a-y too many.
- You may want to control the layer that the text is placed on.
- You may want a specific text style.
- Certainly, you would like some control over the size and orientation of the text.
- Here’s an orthogonal idea: store the x,y,z-coordinates to a file on disk — just in case you ever want to reuse the data.

Conquering Feature Bloat

“Okay,” you may be thinking, “I can agree that these are mostly desirable improvements. Go right ahead, Mr. Grabowski: Show me how to add them in.”

But, wait a minute! When you’re not familiar with LISP, you may not realize how a user interface adds a tremendous amount of code, which mean more bugs and more debugging. (If you are familiar with programming, then you know how quickly a simple program fills up with feature-bloat.) While all those added features sound desirable, they may make the program less desirable. Can you image how irritated you’d get if you had to answer the questions about decimal places, text font, text size, text orientation, layer name, filename — each time you wanted to label a single point?

Take a second look at the wishlist above. Check off features important to you, and then cross out those you could live without.

Wishlist Item #1: Naming the Program

To give the program a name, surround the code with the **defun** function, and give it a name, as follows:

```
(defun c:label ( / xyz)
  (setq xyz (getpoint "Pick point: "))
  (command "text" xyz 200 0 xyz)
)
```

Let's take a look at what's been added, piece by piece:

Defining the Function - defun

(defun defines the name of the function. In LISP, the terms function, program, and routine are used interchangeably (**defun** is short for “define function.”)

Naming the Function - C:

c:label is the name of the function. I decided to call this program “Label”; you can call it anything you like, so long as the name does not conflict with that of any built-in LISP function or other user-defined function. The **c:** prefix makes this LISP routine appear like a progeCAD command.

To run the Label program, all you need do is type “label” at the ‘:’ prompt, like this:

```
Command: label  
Select a point to identify coordinates: (Pick a point.)
```

When the **c:** prefix is missing, however, then you have to run the program like a LISP function, complete with the parentheses, as follows:

```
Command: (label)  
Select a point to identify coordinates: (Pick a point.)
```

Local and Global Variables - /

(/ xyz) declares the names of *input* and *local* variables; the slash separates the two:

- **Input variables** feed data to LISP routines; the names of input variables appear before the slash.
- **Local variables** are used only within programs; the names of local variables appear after the slash.

In this program, **xyz** is the name of the variable that is used strictly within the program. If variables are not declared local, they become *global*. The value of a global variable can be accessed by *any* LISP function loaded into progeCAD.

The benefit to declaring variables as local is that progeCAD automatically frees up the memory used by the variable when the LISP program ends; the drawback is that the value is lost, making debugging harder. For this reason, otherwise-local variables are kept global until the program is debugged.

And the **)** closing parenthesis balances the opening parenthesis at the beginning of the program.

Wishlist Item #2: Saving the Program

By saving the program to a file on disk, you avoid retyping the code with each new progeCAD session. You do this, as follows:

1. Start a text editor (the Notepad supplied with Windows is good).

2. Type the code shown:

```
(defun c:label ( / xyz)
  (setq xyz (getpoint "Pick point: "))
  (command "text" xyz 200 0 xyz)
)
```

I indented the code in the middle to make it stand out from the **defun** line and the closing parenthesis. This is standard among programmers; the indents make it easier to read code. You can use a pair of spaces or the tab key because LISP doesn't care.

3. Save the file with the name *label.lsp* in IntelliCAD's folder.

Wishlist Item #3: Automatically Loading the Program

To load the program into progeCAD, type the following:

```
Command: (load "label")
```

If progeCAD cannot find the LISP program, then you have to specify the path. Assuming you saved *label.lsp* in the `\cad\support` folder, you would enter:

```
Command: (load "\\cad\\support\\label")
```

Now try using the point labelling routine, as follows:

```
Command: label
Select a point to identify coordinates: (Pick a point.)
```

TIP progeCAD provides a way to automatically load LISP programs. When progeCAD starts up, it looks for a file called *icad.lsp*. progeCAD automatically loads the names of LISP programs listed in the file.

Adding *label.lsp* to *icad.lsp* is easy. Open the *icad.lsp* file with a text editor (if the file does not exist, then start a new file called *acad.lsp* and store it in the `\progeCAD` folder). Add the name of the program:

```
(load "label.lsp")
```

Save the *icad.lsp* file. Start progeCAD and it should load *label* automatically.

Wishlist #4: Using Car and Cdr

The x,y,z-coordinates are printed to eight decimal places — that's too many. There are two solutions. One is to ask the user the number of decimal places, as shown by the following code fragment:

```
Command: (setq uprec (getint "Label precision: "))
Label precision: 1
1
```

Or steal the value stored in system variable **LUPrec** — the precision specified by the user through the **Units** command — under the (not necessarily true) assumption that the user want consistent units. The code to do this is as follows:

```
(setq uprec (getvar "LUPREC"))
```

That was the easy part. The tough part is applying the precision to the x,y,z-coordinates, which takes three steps: (1) pick apart the coordinate triplet; (2) apply the precision factor; and (3) join together the coordinates. Here's how:

1. Open *label.lsp* in NotePad or any other text editor. Remove / **xyz** from the code. The variable is now "global," so that you can check its value at progeCAD's ':' prompt. The code should look like this:

```
(defun c:label ( )  
  (setq xyz (getpoint "Pick point: "))  
  (command "text" xyz 200 0 xyz)  
)
```

Save, and then load *label.lsp* into progeCAD.

2. Run *label.lsp*, picking any point on the screen. If you don't see the coordinates printed on the screen, use the Zoom Extents command.
3. At the 'Command:' prompt, enter the following:

```
Command: xyz  
(6.10049 8.14595 10.0)
```

The exclamation mark forces progeCAD to print the value of variable **xyz**, which holds the x,y,z-coordinates. Your results will differ, depending on where you picked.

4. LISP has several functions for picking apart a list. Here you use the **car** and **cdr** functions, and combinations thereof. The **car** function extracts the *first* item (the x-coordinate) from a list. Try it now:

```
Command: (car xyz)  
6.10049
```

5. The **cdr** function is the compliment to **car**. It removes the first item from the list, and then gives you what's left over:

```
Command: (cdr xyz)  
(8.14595 10.0)
```

6. In addition to **car** and **cdr**, LISP allows me to combine the "a" and "d" in several ways to extract other items in the list. To extract the y-coordinate, use **cadr**, as follows:

```
Command: (cadr xyz)  
8.14595
```

7. And to extract the z-coordinate, use **caddr**, as follows:

```
Command: (caddr xyz)  
10.0
```

8. I now have a way to extract the x-coordinate, the y-coordinate, and the z-coordinate from variable **xyz**. I'll store them in their own variables, as follows:

```
Command: (setq ptx (car xyz))  
Missing: 1) > (pty (cadr xyz))  
Missing: 1) > (ptz (caddr xyz))  
Missing: 1) > )
```

You use variable **PtX** to store the x-coordinate, **PtY** for the y-coordinate, and so on. In addition, a form of LISP shorthand was used in the code above that allows you apply the **setq** function to several variables. Recall the reason for progeCAD's 'Missing: 1) >' prompt: it reminds you that a closing parenthesis is missing.

- Now that the three coordinates are separated, you can finally reduce the number of decimal places. There are a couple of ways to do this. Use the **rtos** function, because it does two things at once: (1) changes the number of decimal places to any number between 0 and 8; and (2) converts the real number into a string. Why a string? You'll see later. For now, here is the **rtos** function at work:

```
Command: (rtos ptx 2 uprec)
"6.1"
```

The **rtos** function uses three parameters:

PtX Name of the variable holding the real number.
2 Type of conversion, decimal in this case. The number **2** is based on system variable **LUnits**, which defines five modes of units:

Mode	Units
1	Scientific
2	Decimal
3	Engineering
4	Architectural
5	Fractional

UPrec Name of the variable holding the precision (the code for that is at the beginning of this section). This varies, depending on the type of units. For example, a value of 2 for decimal means two decimal places; a 2 for architectural means quarter-inch.

Assuming, then, that the precision in **UPrec** is 1, the **rtos** function in the code fragment above reduces 6.10049 to 6.1.

- Truncate, and preserve the values of x, y, and z three times, as follows:

```
Command: (setq ptx (rtos ptx 2 uprec)
1> pty (rtos pty 2 uprec)
1> ptz (rtos ptz 2 uprec)
1> )
```

Notice that you can set a variable equal to itself: **PtX** holds the new value of the x-coordinate after **rtos** gets finished processing the earlier value stored in **PtX**. Reusing a variable name like this helps conserve memory.

- With the coordinates truncated, you now have to string (pardon the pun) them together with the **strcat** function, short for string concatenation. Try it now:

```
Command: (strcat ptx pty ptz)
"6.18.110.0"
```

- Oops!* Not quite the look you may have been hoping for. Since LISP can't know when you want spaces, it provides none. You have to insert them yourself using **strcat**, one of the most useful LISP functions. It lets you create a string that contains text and variables, like this:

```
Command (setq xyz (strcat ptx ", " pty ", " ptz))
"6.1, 8.1, 10.0"
```

That's more like it!

13. Back to the text editor. Add in the code you developed here, shown in **boldface**, and with LISP functions in **cyan**:

```
(defun c:label (/ xyz xyz1 uprec ptx pty ptz)
  (setq uprec (getint "Label precision: "))
  (setq xyz (getpoint "Pick point: "))
  (setq ptx (car xyz)
        pty (cadr xyz)
        ptz (caddr xyz)
  )
  (setq ptx (rtos ptx 2 uprec)
        pty (rtos pty 2 uprec)
        ptz (rtos ptz 2 uprec)
  )
  (setq xyz1 (strcat ptx ", " pty ", " ptz))
  (command "text" xyz 200 0 xyz1)
)
```

Notice that all variables are local. Notice, too, the change to variable **xyz** in the last couple of lines: you don't want the text placed at the rounded-off coordinates, so use **xyz1** as the variable holding the text string.

14. Finally, you should add comments to your code to remind you what it does when you look at the code several months from now. Semicolons indicate the start of comments:

```
; Label.Lsp labels a picked point with its x,y,z-coordinates.
; by Ralph Grabowski, 25 February, 1996.
(defun c:label (/ xyz xyz1 uprec ptx pty ptz)
  ; Ask user for the number of decimal places:
  (setq uprec (getint "Label precision: "))
  ; Ask the user to pick a point in the drawing:
  (setq xyz (getpoint "Pick point: "))
  ; Separate 3D point into individual x,y,z-values:
  (setq ptx (car xyz)
        pty (cadr xyz)
        ptz (caddr xyz)
  )
  ; Truncate values:
  (setq ptx (rtos ptx 2 uprec)
        pty (rtos pty 2 uprec)
        ptz (rtos ptz 2 uprec)
  )
  ; Recombine individual values into a 3D point:
  (setq xyz1 (strcat ptx ", " pty ", " ptz))
  ; Place text:
  (command "text" xyz 200 0 xyz1)
)
```

15. Save the file as *label.lsp*, then load the LISP routine into progeCAD with:

```
Command: (load "label")
"C:LABEL"
```

16. Run the routine, and respond to the prompts:

```
Command: label
Label precision: 1
Pick point: (Pick a point.)
text Justify.../⟨Start point⟩:
Height of text <200.0000>: 200
Rotation angle of text <0>: 0
Text: 5012.3, 773.2, 0.0
Command:
```

Saving Data to Files

In the previous tutorial, we begin to worry about user interface enhancements. What started out as two lines of code has now bulged out into 23. In this tutorial, we learn how to fight feature bloat (more later), and add the ability to save data to a file.

A reader wrote me with this wishlist item: “The LISP file comes in very handy with some of the programs I use, but I would like to be able to save the data collected on the x,y,z-coordinates in a text file.”

Saving the data to file is easily done with the **open**, **write-line**, and **close** functions. Let’s take a look at how to do this. Dealing with files in LISP is simpler than for most programming languages because LISP has very weak file access functions. All it can do is read and write ASCII files in sequential order; LISP cannot deal with binary files nor can it access data in random order.

The Three Steps

There are three steps in writing data to a file:

- Step 1. Open** the file.
- Step 2. Write** the data to the file.
- Step 3. Close** the file.

Step 1: Open the File

LISP has the **open** function for opening files. The function lets you open files for *one* of three purposes: (1) read data from the file; (2) write data to the file; or (3) append data to the file. You must choose one of these at a time; LISP cannot do all three at once.

In all cases, LISP takes care of creating the file if it does not already exist. Reading data is easy enough to understand, but what’s the difference between “writing” and “appending” data?

- When I ask progeCAD to open a file to **write**, all existing data in that file is *erased*, and then the new data is added.
- When I ask progeCAD to open a file to **append**, the new data is *added* to the end of the existing data.

For our purpose, we want to keep adding data to the file, so choose *append* mode. The LISP code looks like this:

```
(setq FIL (open "xyzdata.txt" "a"))
```

Here you are setting something (through **setq**) equal to a variable named **FIL**. What is that? In pretty much all programming languages, we don't deal with the file name directly, but instead deal with a *file descriptor*. This is a name (some sequence of letters and numbers) to which the operating system assigns the file name. Now that you have the file descriptor stored in variable **FIL**, you work with **FIL**, not the file name, which I have decided to call *xyzdata.txt*. The final "a" tells LISP you want to open *xyzdata.txt* for appending data. It is important that the "a" be lowercase; this is one of the very few occasions when LISP is case-sensitive. The options for the **open** function are:

Option	Comment
"a"	Append data to end of file.
"w"	Write data to file (erase existing data).
"r"	Read data from file.

Step 2: Write Data to the File

To write data to files, use the **write-line** function. This function writes one line of data at a time. (Another function, the **write** function, writes single *characters* to files.) The code looks like this:

```
(write-line xyz1 fil)
```

You cannot, however, just write raw data to the file because it would look like three decimal points and a lot of digits, like this:

```
8.15483.27520.0000
```

Most software is able to read data with commas separating numbers, like this:

```
8.1548, 3.2752, 0.0000
```

That includes spreadsheets, database programs, and even some word processing software. I tell these programs that when they read the data, they should consider the comma to be a *separator* and not a comma. In that way, the spreadsheet program places each number in its own cell. With each number in its own cell, I can manipulate the data. For this reason, you need code that formats the data.

Fortunately, you've done that already. Last tutorial, you used the **strcat** function along with the **cdr**, **cadr**, and **caddr** functions to separate the x, y, and z components of the coordinate triplet. So you can reuse the code, which looks like this:

```
(setq ptx (car xyz)
      pty (cadr xyz)
      ptz (caddr xyz)
)
(setq xyz1 (strcat ptx ", " pty ", " ptz))
```

The **strcat** function places the commas between the coordinate values.

Step 3: Close the File

Finally, for good housekeeping purposes, close the file. progeCAD will automatically close the file for you if you forget, but a good programmer cleans up after themselves. Closing the file is as simple as:

```
(close fil)
```

Putting It Together

Add the code for opening, formatting, writing, and closing to the *lable.lsp* program:

```
(defun c:label (/ xyz xyz1 uprec ptx pty ptz)
  (setq uprec (getint "Label precision: "))
  (setq xyz (getpoint "Pick point: "))
  (setq ptx (car xyz)
        pty (cadr xyz)
        ptz (caddr xyz)
  )
  ; Format the x,y,z coordinates:
  (setq ptx (rtos ptx 2 uprec)
        pty (rtos pty 2 uprec)
        ptz (rtos ptz 2 uprec)
  )
  ; Add commas between the three coordinates:
  (setq xyz1 (strcat ptx ", " pty ", " ptz))
  ; Write coordinates to the drawing:
  (command "text" xyz 200 0 xyz1)
  ; Open the data file for appending:
  (setq fil (open "xyzdata.txt" "a"))
  ; Write the line of data to the file:
  (write-line xyz1 fil)
  ; Close the file:
  (close fil)
)
```

Using a text editor, such as Notepad, make the additions (shown in **boldface** above) to your copy of *lable.lsp*. Load it into progeCAD with the **load** function:

```
Command: (load "label")
```

And run the program by entering **Label** at the ':' prompt:

```
Command: label
Label precision: 4
Pick point: (Pick a point.)
```

As you pick points on the screen, the routine labels the picked points, but also writes the 3D point data to file. After a while, this is what the data file looks something like this:

```
8.1548, 3.2752, 0.0000
7.0856, 4.4883, 0.0000
6.4295, 5.6528, 0.0000
5.5303, 6.7688, 0.0000
5.4331, 8.3215, 0.0000
```

Wishlist #5: Layers

Let's take a moment to revisit the wishlist. One wishlist item is to control the layer on which the text is placed. There are two ways to approach this wishlist item:

- The no-code method is to set the layer before starting the LISP function.
- The LISP-code version is to ask the user for the name of the layer, then use the **setvar** function to set system variable **CLayer** (much easier than using the **Layer** command), as follows:

```
(setq lname (getstring "Label layer: "))  
(setvar "CLAYER" lname)
```

Add those two line before the line with the "Pick point" prompt.

Wishlist #6: Text Style

To specify the text style, there are the same two methods. The no-code method is to simply set the text style before starting the routine. Otherwise, you can write LISP code similar to set the style with the **setvar** command, as follows:

```
(setq tname (getstring "Label text style: "))  
(setvar "TEXTSTYLE" tname)
```

Once again, add those two line before the line with the "Pick point" prompt.

By now, you may be noticing that your program is starting to look big. This is called "feature bloat." More features, especially in the area of user interface, makes software grow far beyond the size of its basic algorithm.

Tips in Using LISP

To conclude this chapter, here are tips for helping out when you write your LISP functions.

Tip #1. Use an ASCII Text Editor.

LISP code must be written in plain ASCII text — no special characters and no formatting (like **boldface** or **color**) of the sort that word processors add to the file. When you write LISP code with, say, Word, then save as a *.doc*-format file (the default), progeCAD will simply refuse to load the LISP file, even when the file extension is *.lsp*.

In an increasingly Window-ized world, it is harder to find a true ASCII text editor. There is one, however, supplied free by Microsoft with Windows called Notepad. Do not use Write or WordPad supplied with Windows. While both of these have an option to save in ASCII, you're bound to forget sometimes and end up frustrated.

Almost any other word processor has an option to save text in plain ASCII, but not by default. Word processors have a number of different terms for what I mean by "pure ASCII format." Word calls it "Text Only"; WordPerfect calls it "DOS Text"; WordPad calls it "Text Document"; and Atlantis calls it "Text Files." You get the idea.

Tip #2: Loading LSP Code into progeCAD

To load the LISP code into progeCAD, you use the **load** function. Here's an example where *points.lsp* is the name of the LISP routine:

```
Command: (load "points")
```

You don't need to type the *.lsp* extension.

When progeCAD cannot find *points.lsp*, you need to specify the folder name by using either a forward slash or double backslashes — your choice:

```
Command: (load "\\progeCAD\\points")
```

After you've typed this a few times, you'll find it gets tedious. To solve the problem, write a one-line LISP routine that reduces the keystrokes, like this:

```
Command: (defun cx 0 (load "points"))
```

Now anytime you need to load the *points.lsp* routine, you just type **X** and press **Enter**, as follows:

```
Command: x
```

Under Windows, you could also just drag the *.lsp* file from the File Manager into progeCAD. Note that the code moves one way: from the text editor to progeCAD; you cannot drag the code from progeCAD back to the text editor.

Tip #3: Toggling System Variables

One problem in programming is: How to change a value when you don't know what the value is? In progeCAD, you come across this problem with system variables, many of which are toggles. A *toggle* system variable has a value of 0 or 1, indicating that the value is either off (0) or on (1). For example, system variable **SpIframe** is by default 0: when turned off, splined polylines do not display their frame.

No programmer ever assumes that the value of **SpIframe** is going to be zero just because that's its default value. In the case of toggle system variables, there two solutions:

- (1) Employ the **if** function to see if the value is 0 or 1.
- (2) Subtract 1, and take the absolute value.

Tip #4: Be Neat and Tidy.

Remember, your mother told you to always pick up your things. This problem of setting system variables applies universally. When your LISP routine changes values of system variables, it must always set them back to the way they were before the routine began running.

Many programmers write a set of generic functions that save the current settings at the beginning of the routine, carries out the changes, and then restores the saved values at the end of the routine. Here's a code fragment that shows this, where the original value of **SpIframe** is stored in variable **SpIvar** using **getvar**, and then restored with **setvar**:

```
(setq splvar (getvar "splframe"))
```

```
...
```

```
(setvar "splframe" splvar)
```

Tip #5: UPPER vs. lowercase

In (almost) all cases, LISP doesn't care if you use UPPERCASE or lowercase for writing the code. For legibility, there are some conventions:

- LISP function names in all lowercase.
- Your function names in Mixed Case.
- progeCAD variables and command names in all UPPERCASE.

As I said, LISP doesn't care, and converts everything into uppercase in any case. It also strips out all comments, excess white space, tabs, and return characters. The exception is text in quote marks, such as prompts, which are left as is.

There are two exception where LISP does care: when you are working with file functions, escape codes, and the letter T. The **open** function uses the arguments "**r**", "**w**", and "**a**" to read to, write from, and append to a file, respectively. Those three characters must be lowercase.

Escape codes are used in text strings, and must remain lowercase. For example, `\e` is the escape character (equivalent to ASCII 27) and `\t` is the tab character. Note that they use backslashes; it is for this reason that you cannot use the backslash for separating folders names back in Tip #2. LISP would think you were typing an escape code.

And some functions use the letter T as a *flag*. It must remain uppercase.

Tip # 6: Quotation Marks as Quotation Marks

As we have seen, LISP uses quotation marks `"` and `'` for strings. Thus, you cannot use a quotation mark as for displaying quotation marks and inches, such as displaying 25 inches as 25".

The workaround is to use the escape codes mentioned above in Tip #5, specifically the octal code equivalent for the ASCII character for the quotation mark. Sound complicated? It is. But all you need to know is 042. Here's how it works:

First, assign the strings to variables, as follows:

```
(setq distxt "The length is ")  
(setq distval 25)  
(setq qumark "\042")
```

Notice how I assigned octal 042 to variable **qumark**. The backslash tells LISP the numbers following are in octal. Octal, by the way, is half of hexadecimal: 0 1 2 3 4 5 6 7 10 11 12 ... 16 17 20 21 ...

Then concatenate the three strings together with the **strcat** function:

```
(strcat distxt distval qumark)
```

To produce the prompt:

```
The length is 25"
```

Tip #7: Tabs and Quotation Marks

Vijay Katkar is writing code for a dialog box with a list box. He told me, “I want to display strings in it — just like the dialog box displayed by the **Layer** command. I am able to concatenate the values and print the strings but there is no vertical alignment, since the strings are of different lengths. I tried using the tab metacharacter (`\t`) in the string but it prints the literal `'\t'` in the list box. Is there any way I can get around this problem?”

I recall a similar problem: How to display quotation marks or the inches symbol within a text string? For example, I have a line of LISP code that I want to print out as:

```
The diameter is 2.54"
```

Normally, I cannot use the quotation (`"`) character in a string. LISP uses the quotation as its string delimiter to mark the beginning and ending of the string. In the following line of code:

```
(prompt "The diameter is 2.54"")
```

LISP sees the first quotation mark as the start of the string, the second quotation as the end of the string, and the third quotation mark as an error.

The solution is the `\nnn` metacharacter. This lets me insert any ASCII character, including special characters, such as tab, escape, and quotation marks. The workaround here is to use the ASCII code for the quotation mark, `\042`, like this:

```
(prompt "The diameter is 2.54\042")
```

Similarly, Vijay needs to use the `\009` metacharacter to space the text in his dialog box. And, in fact, that worked: “According to what you had told me, I used the same and it worked.”

Introduction to DCL

dCL allows programmers to create custom dialog boxes. (DCL is short for “dialog control language.”) Indeed, some of progeCAD’s dialog boxes are written in DCL, rather than being hard-coded; look for *.dcl* files in progeCAD’s folders.

DCL is a structured language that describes all the elements (called “tiles”) that make up dialog boxes: edit boxes, list boxes, radio buttons, image tiles, and so on. Each tile has one or more attributes, such as its position, background color, and the action it performs. Applications written in LISP, SDS, and DRX can make use of DCL for dialog boxes.

Autodesk provides no programming environment to help you create DCL files — it’s hand coding all the way. That means the Notepad text editor becomes your DCL programming environment. Some third-party developers created DCL development tools. (If you program with VBA, then DCL is not needed.)

When working with DCL and LISP, you need to write two pieces of code:

- *.dcl* files that define the dialog box and the function of its tiles.
- *.lsp* files that load the *.dcl* files, and activate the tiles.

TIP DCL is still under development in progeCAD, and so not all of this chapter’s examples will necessarily work.

In This Chapter

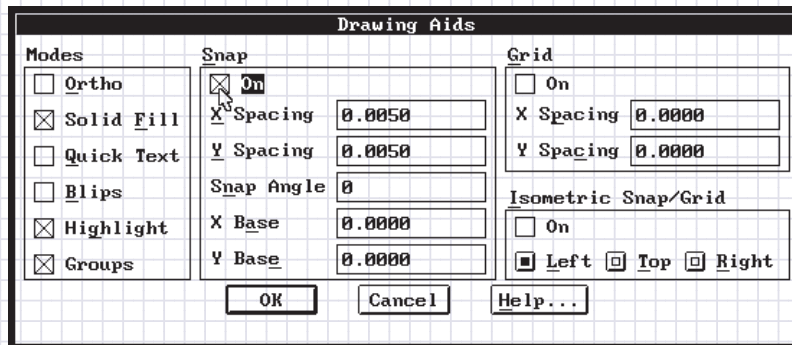
- History of DCL
- Writing and testing DCL code
- What dialog boxes are made of
- LISP code to run DCL

History of DCL

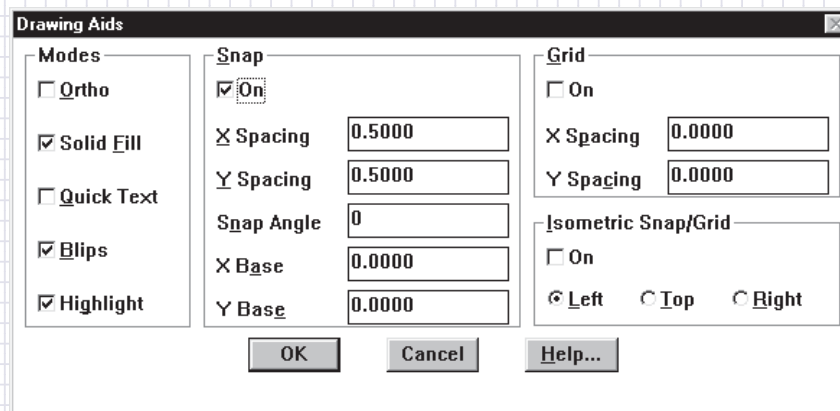
Autodesk first introduced DCL (short for “dialog control language”) in AutoCAD Release 11 for Windows as an undocumented feature. DCL is a syntax for creating platform-independent dialog boxes.

At that time, Autodesk was producing versions of AutoCAD for “every viable engineering platform,” which included DOS, Windows, Unix, Macintosh, and OS/2. DCL was part of a project code-named “Proteus,” whose aim was to make AutoCAD work and look identical on every operating system.

As the two figures below show, the project was a success. The DOS and Windows dialog boxes look very similar. Here is AutoCAD Release 11’s Drawing Modes dialog box running on DOS:



And here is the same dialog box in AutoCAD Release 11 for Windows:



By Release 14, however, Proteus was meaningless, because Autodesk chose to support only the Windows operating system. But DCL still hangs around as the only way to create dialog boxes with AutoLISP.

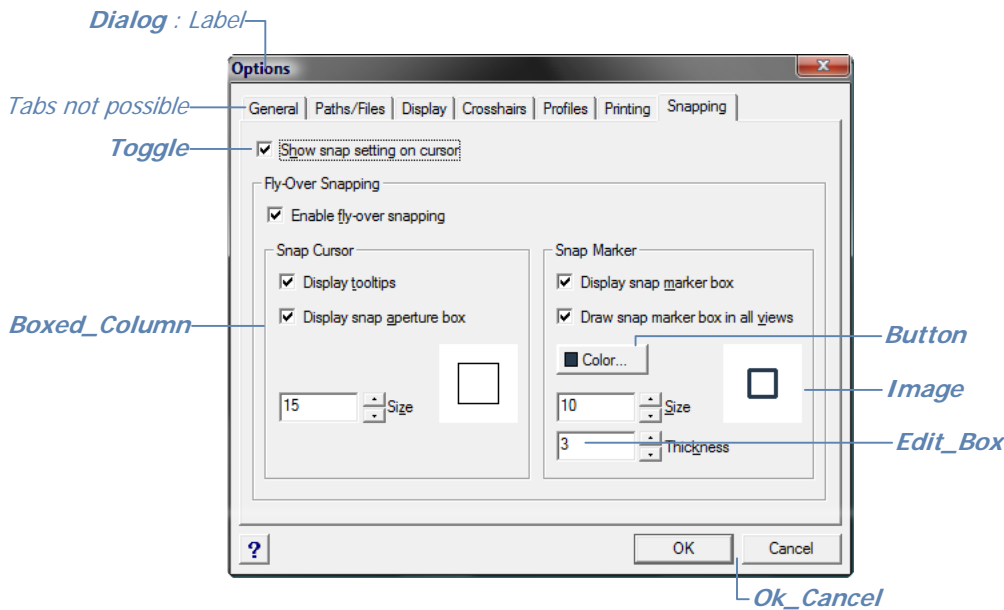
More recently, the Open Design Alliance added DCL to its offerings, and members of the IntelliCAD Technical Consortium adopted it, including progeCAD.

Thus, working with dialog boxes always involves a pair of files, *.dcl* and *.lsp*. And it's the LISP code that controls the dialog box code.

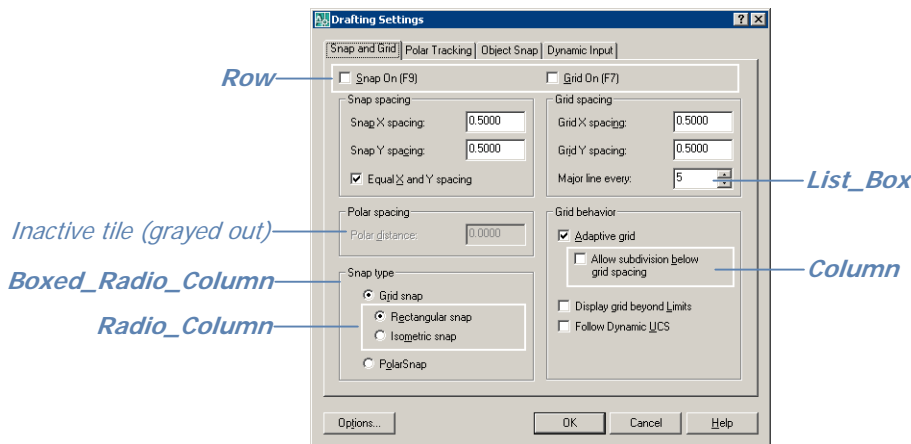
A drawback to DCL is that it cannot self-modify dialog boxes, such as adding or removing buttons.

What Dialog Boxes Are Made Of

Dialog boxes can consist of many elements, such as radio buttons, sliders, images, tabs, and check boxes. (These elements are called “tiles.”) DCL allows you to create many elements — but not all. The figure below illustrates some of the dialog box elements that are possible with DCL. The boldface text names the specific DCL tiles. (Those tiles not possible with DCL can be created through VBA, which is not discussed in this ebook.)



Some tiles are invisible, such as some of the ones highlighted below by white rectangles:



Two pieces of code are always required to make dialog boxes operate: (1) DCL code specifies the layout of tiles and their attributes in the dialog box; and (2) LSP code activates the dialog box. progeCAD automatically sizes dialog boxes for you, and stacks tiles in columns by default.

Some back and forth is permitted while running DCL and LISP files; this is called a “callback.” For example, the file name “Drawing1.dwg” is inserted in the dialog box by the LISP **getvar** function calling system variable **DwgName**. Callbacks are also used to gray out buttons and to change the content of popup lists (droplists).

This chapter shows how to write DCL and LISP code. The next chapter provides you with a comprehensive reference to all DCL tiles, their attributes, and related LISP functions.

Your First DCL File

Before writing code for dialog boxes, it is helpful to plan out the tiles. Where in the dialog box will the buttons, droplists, and text entry boxes go? The best thing is to get a pencil, and then sketch your idea on paper.

For this tutorial, we create a dialog box that displays the values stored in these system variables:

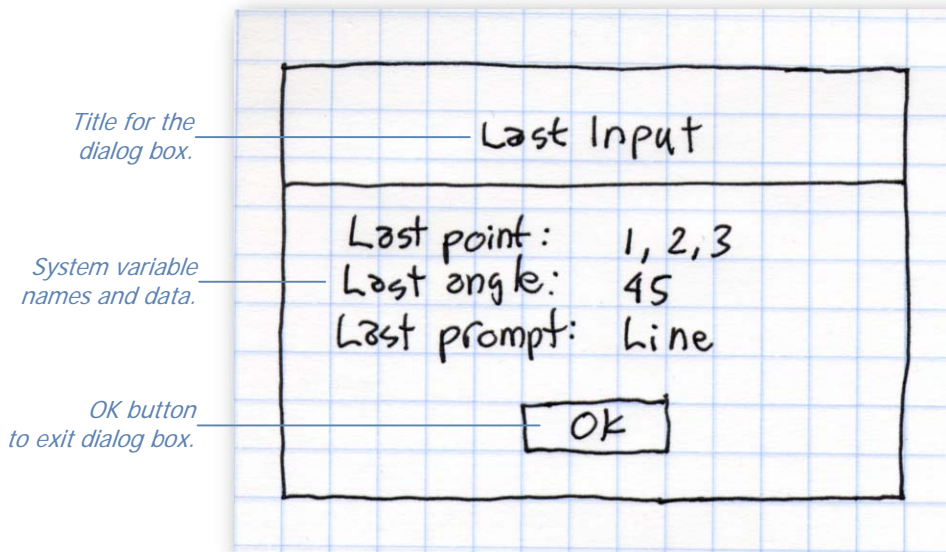
LastPoint — the last 3D point entered in the drawing.

LastAngle — the angle defined by the last two points entered.

LastPrompt — the last text entered by the user at the command line.

Think about how the dialog box would be constructed. It would probably have three lines of text reporting the name and value of each system variable. It would have an **OK** button to exit the dialog box. And it might have a title that explains the purpose of the dialog box.

It might look like this:



DCL Metacharacters

Meaning

//	(slash-slash) Indicates a comment line.
/*	(slash-asterisk) Starts comment section.
*/	(asterisk-slash) Ends comment section.
:	(colon) Starts a tile definition. ¹
{	(brace) Starts dialog and tile attributes.
	(space) Separates symbols.
=	(equals) Defines attribute values.
""	(quotations) Encloses text attributes.
;	(semi-colon) Ends attribute definition. ²
}	(brace) Ends tile and dialog attributes.

Notes:

¹ Predefined tiles, such as spacer, are not prefixed with the colon.

² Every attribute must end with a semi-colon.

TIPS To grab the screen image of dialog boxes, hold down the **Alt** key, and then press **Prt Scr**. That captures just the dialog box to the Clipboard. Switch to a paint program, such as PaintShop Pro, and then press **Ctrl+V** to paste the image. (This is the method I use in this ebook.)

The DCL and LSP files that you create for AutoCAD can also be used with IntelliCAD. Note that there may be some differences, because IntelliCAD does not support all DCL and AutoLISP functions and their parameters. Some of these differences are documented by Adeko of Greece at www.adeko.com.tr/forum/topic.asp?TOPIC_ID=1263 and by IcadWeb of Japan at www.icadweb.com/doc/help/dr/IDR_DCL_B1.htm.

For example, IntelliCAD adds these attributes to the **edit_box** tile:

typeface	specifies the TrueType font.
pointsize	specifies the size of text.
read_only	prevents text from being edited.
lower_only	reads text as lower case.
upper_only	reads text as all uppercase.

DCL Programming Structure

The programming structure of the dialog box might look like this:

```
Start dialog box definition.  
Dialog box title.  
Column of system variable names and data.  
OK button.  
End of dialog box.
```

In this first tutorial, we will first write just enough code to display the dialog box and the OK button. Later, we add bells and whistles.

Start Dialog Box Definition

The `.dcl` files that define dialog boxes begin with a name. This is the name by which the code is later on called by the associated LISP routine.

```
name: dialog {
```

Like LISP, the open brace needs a closing brace to signal the end of the dialog box definition:

```
}
```

For this tutorial, let's call it "lastInput," as follows:

```
lastInput: dialog {  
}
```

The name is case-sensitive, so "lastInput" is not the same to LISP as "LastINPUT" or "lastinput."

Dialog Box Title

The label property gives the dialog box its title.

```
name: dialog {  
  label = "Dialog box title";  
}
```

For this tutorial, label it "Last Input," as follows:

```
lastInput: dialog {  
  label = "Last Input";  
}
```

The title text needs to be surrounded by quotation marks. The label property must be terminated with the semi-colon. And it's helpful to indent the code to make it readable.

OK Button

Every dialog box needs an exit button, either OK or Cancel — otherwise there is no way to exit it. If no button, then progeCAD refuses to run the DCL code, and instead displays a warning.

Before we test our DCL code for the first time, we need to include code for the OK button. It is shown in color:

```
lastInput: dialog {  
  label = "Last Input";  
  : button {  
    key = "okButton";  
    label = " OK ";  
    is_default = true;  
  }  
}
```

Buttons are defined with the **button** property; the properties of the button are enclosed in braces:

```
: button {  
}
```

Because dialog boxes can have multiple buttons, every button must be identified by the “key” property. The key allows LISP to give the button its instructions later on. Identify this OK button as “okButton” with the **key** attribute, as follows:

```
key = "okButton";
```

The button needs to display a label for users. This is the OK button, so label it “OK” with the **label** attribute, as follows:

```
label = " OK ";
```

To make life easier for users, one tile of the dialog box is made the *default*. Dialog boxes highlight the default tile in some way, such as the dashed outline of the OK button. When a button is the default, users need only press **Enter** to activate it. Make this button the default one with the **is_default** attribute, as follows:

```
is_default = true;
```

TIP This DCL code for the **OK** button is like a subroutine. Its code can be used and reused any time a dialog box needs an **OK** button, which is pretty much every time. Later, we see how to create subroutines in DCL code.

```
: button {  
  key = "okButton";  
  label = " OK ";  
  is_default = true;  
}
```

Basic LISP Code to Load and Run Dialog Boxes

The following LISP code loads, runs, and exits the *lastInput.dcl* file:

```
(defun C:xx ()  
  (setq dlg-id (load_dialog "c:\\lastInput"))  
  (new_dialog "lastInput" dlg-id)  
  (action_tile "accept" "(done_dialog)")  
  (start_dialog)  
  (unload_dialog dlg-id)  
)
```

Let's take apart the code to see how it operates.

```
(defun C:xx ()
```

The function is defined as "xx" with LISP's **defun** function. *Programming = debugging*, so I like to use an easy-to-enter name for the LISP routine, like "xx."

```
(setq dlg-id (load_dialog "c:\\lastInput"))
```

The *lastInput.dcl* file is loaded with the **load_dialog** function. There is no need to specify the ".dcl" extension, because this function's sole purpose is to load DCL files. Recall that LISP requires you to use \\ instead of \ for separating folder names.

```
(new_dialog "lastInput" dlg-id)
```

DCL files can contain more than one dialog box definition, so the next step is to tell progeCAD which one you want with the **new_dialog** function. In this case, we have just the one, "lastInput."

```
(action_tile "okButton" "(done_dialog)")
```

Our dialog box contains a button named "okButton," and its purpose is defined by LISP — not DCL! Here we use the **action_tile** function to assign the "okButton" button its purpose in life: to execute the **done_dialog** function that exits the dialog box. You can read this as "the *action* for the *tile* named *okButton* is ...". In short, click **OK** to exit the dialog box. In place of "(done_dialog)" you can use any AutoLISP code you like.

```
(start_dialog)
```

After all these preliminaries, the big moment arrives. The **start_dialog** function launches the dialog box, and waits then for you to click its button.

```
(unload_dialog dlg-id)
```

As neat programmers, we unload the dialog box from memory with the **unload_dialog** function.

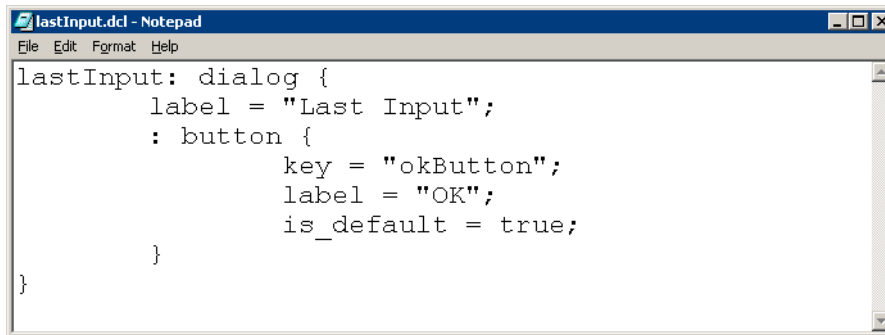
```
)
```

And a final parenthesis ends the *xx* function.

Testing the DCL Code

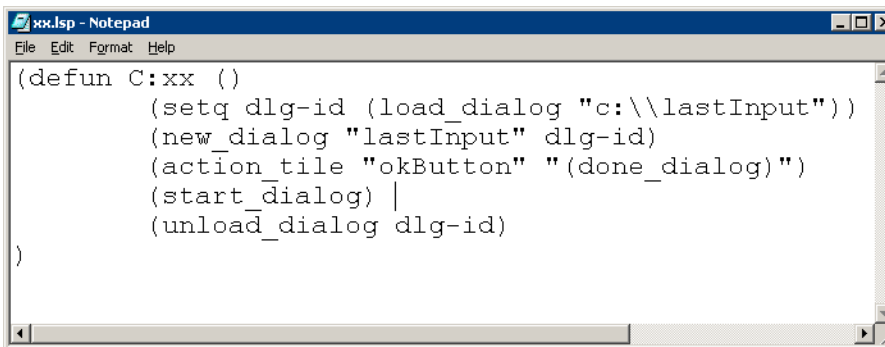
We have enough DCL code to test it, and see how the dialog box is developing. To test the code, take these steps:

1. Open Notepad or another ASCII text editor.
2. Enter the code we developed earlier:



```
lastInput: dialog {
  label = "Last Input";
  : button {
    key = "okButton";
    label = "OK";
    is_default = true;
  }
}
```

3. Save the file as *lastinput.dcl* to the C:\ drive, so that the LISP *xx.lsp* routine can find it.
4. Enter the LISP code described on the previous page:

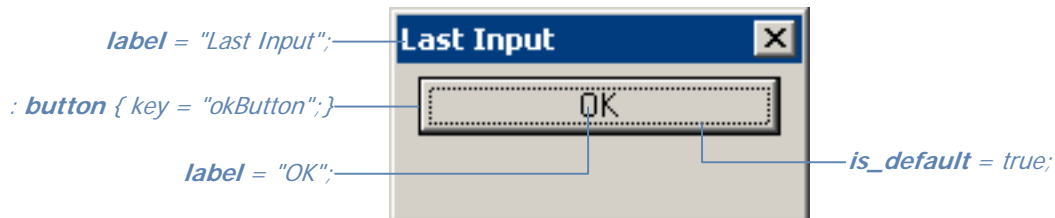


```
(defun C:xx ()
  (setq dlg-id (load_dialog "c:\\\\lastInput"))
  (new_dialog "lastInput" dlg-id)
  (action_tile "okButton" "(done_dialog)")
  (start_dialog) |
  (unload_dialog dlg-id)
)
```

5. Save the file as *xx.lsp*, also on the C:\ drive.
6. Switch to progeCAD, and drag the *xx.lsp* file from Windows Explorer into the drawing window. This is much easier than using the AppLoad command or the load function!
7. Type **xx** to run the routine and load the dialog box:

Command: **xx**

Notice that the dialog box appears. Click **OK** to exit it.



Displaying System Variable Data

The basic structure of the dialog box is in place: the label and the OK button. Let's add the data displayed by the system variables. They will look something like this:

```
Last angle:      45
Last point:     1,2,3
Last prompt:    Line
```

The colored text is static, and acts like a prompt. The italicized text is variable; its display depends on the value of the associated variable.

The **Text** tile displays text in dialog boxes, as follows:

```
: text {
  label = "Last angle: ";
  key = "lastAngle";
}
```

Are you able to recognize the attributes of this text tile?

```
: text {
```

...begins the text tile.

```
  label = "Last angle: ";
```

The label attribute provides the prompt, 'Last angle: '.

```
  key = "lastAngle";
```

The key attribute identifies the text tile as "lastAngle."

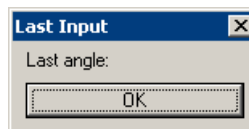
```
}
```

And the text tile is closed with the brace.

TIP Text tiles can have the following attributes, as described fully in the DCL reference later in this ebook:

- alignment
 - fixed_height
 - fixed_width
 - height
 - is_bold
 - key
 - label
 - value
 - width
-

Load and run the DCL code; it displays the 'Last angle:' text:



The next step is to add the display of the value stored by the LastAngle system variable.

Add a second text tile:

```
: text {  
  value = "";  
  key = "lastAngleData";  
}
```

This one is initially blank, because it has no label and no value. To complete the text tile, we need a LISP function that extracts the value from the **LastAngle** system variable, and shows it in the dialog box.

The DCL file now looks like this, with the new code shown in color:

```
lastInput: dialog {  
  label = "Last Input";  
  : text {  
    label = "Last angle: ";  
    key = "lastAngle";  
  }  
  : text {  
    value = "";  
    key = "lastAngleData";  
  }  
  : button {  
    key = "okButton";  
    label = "OK";  
    is_default = true;  
  }  
}
```

Adding the Complimentary LISP Code

Writing DCL code is always only half the job. The other half is to write the complementary code in AutoLISP. To extract the value from LastAngle, we take two steps:

Step 1: Use the **getvar** function, and then store (**setq**) the gotten value in variable *lang* (short for “last angle”), as follows:

```
(setq lang (getvar "LastAngle"))
```

Step 2: Use the **set_tile** function to set the value of *lang* to the “lastAngleData” tile:

```
(set_tile "lastAngleData" (rtos lang 2 2))
```

TIP Tiles can only work with text. The value of **LastAngle** is a *real* number (contains a decimal), so we have to convert it to text with the **rtos** function:

(rtos lang 2 2)

This converts the real number to a string (a.k.a. text) using mode 2 (decimal) and precision 2 (two decimal places).

With the new lines of code shown in color, the LSP file now looks like this:

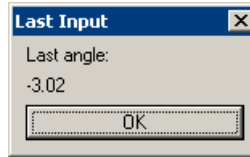
```
(defun C:xx ()  
  (setq dlg-id (load_dialog "c:\\lastInput"))  
  (new_dialog "lastInput" dlg-id)  
  (setq lang (getvar "lastangle"))  
  (set_tile "lastAngleData" (rtos lang 2 2))  
)
```

```

(action_tile "okButton" "(done_dialog)")
(start_dialog)
(unload_dialog dlg-id)
)

```

Save the *.dcl* and *.lsp* files, and then reload and rerun *xx.lsp* in progeCAD. The dialog box looks like this:



Clustering Text

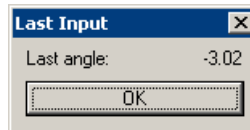
Hmmm... that stacked text is a problem. The solution is to cluster the two text tiles so that they appear next to each other. This is done with the **Row** tile, as shown in color:

```

: row {
  : text {
    label = "Last angle: ";
    key = "lastAngle";
  }
  : text {
    value = "";
    key = "lastAngleData";
  }
}

```

Modify the DCL file, and then rerun the LSP file. The result should look better, like this:



With the last angle text looking proper, we can copy and paste its code, and then make suitable modifications to create the other two lines of text. The changes are shown in color, first for the DCL code for the last point:

```

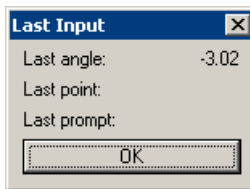
: row {
  : text {
    label = "Last point: ";
    key = "lastPoint";
  }
  : text {
    value = "";
    key = "lastPointData";
  }
}

```

And then for last prompt:

```
: row {  
  : text {  
    label = "Last prompt: ";  
    key = "lastPrompt";  
  }  
  : text {  
    value = "";  
    key = "lastPromptData";  
  }  
}
```

Running *xx.lsp* gives us all three prompts, but we still need to fill in the data for two of them:



Supplying the Variable Text

The data is supplied by LISP code. Recall that LISP returns the value of points as list of three numbers, such as:

```
(1.0000 2.0000 3.0000)
```

The numbers represent the x, y, and z coordinate, respectively. To convert the list to three real numbers represented as a string — why does it have to be so hard?! — use the following code. This assumes that *lpt* contains (1.0000 2.0000 3.0000):

```
(car lpt)
```

The **car** function extracts the x-coordinate from the list as a real number, such as 1.0000. Similarly:

```
(cadr lpt)  
(caddr lpt)
```

The **cadr** and **caddr** functions extract the y (2.0000) and z (3.0000) coordinates, respectively. To convert the real numbers to strings, use the **rtos** function, as follows:

```
(rtos (car lpt))  
(rtos (cadr lpt))  
(rtos (caddr lpt))
```

And then to convert the three individual strings to one string, use the **strcat** (string concatenation) function, as follows:

```
(strcat  
  (rtos (car lpt))  
  (rtos (cadr lpt))  
  (rtos (caddr lpt))  
)
```

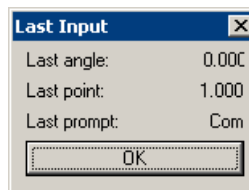

This will result in a display of 1.000 2.000 3.000. It would be nice to put commas between the numbers:

```
(strcat
  (rtos (car lpt) "")
  (rtos (cadr lpt) "")
  (rtos (caddr lpt)
)
)
```

Putting both lines of code together, we arrive at the LISP needed to implant the value of the LastPoint system variable in the dialog box:

```
(setq lpt (getvar "lastpoint"))
(set_tile "lastPointData" (strcat (rtos (car lpt) "") (rtos (cadr lpt) "") (rtos (caddr lpt))))
```

Add the code to the xx.lsp, and then run it in progeCAD to see the result.



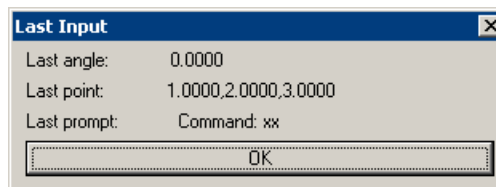
Leaving Room for Variable Text

Oops, the text is cut off. The dialog box is sized before the LISP code inserts the data, so it doesn't know that the dialog box needs to be bigger to accommodate the x,y,z coordinates — which can run to many characters in length.

The solution is to use the width attribute for three of the text tiles, like this:

```
: text {
  value = "";
  key = "lastAngleData";
  width = 33;
}
```

When added to the DCL file, the result looks like this:

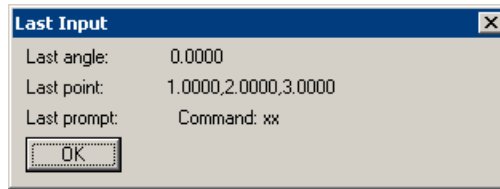


Fixing the Button Width

The other thing that looks wrong to me is the width of the OK button. To make it narrower (i.e., fix its width), use the **fixed_width** attribute in the DCL file:

```
fixed_width = true;
```

By setting it to true, the button is only as wide as it needs to be.



Centering the Button

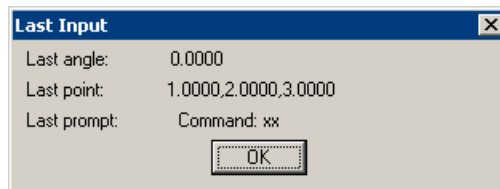
The button is right-justified by default; use the **alignment** attribute to center the button in the dialog box:

```
alignment = center;
```

Add the new code to the button portion of the DCL file...

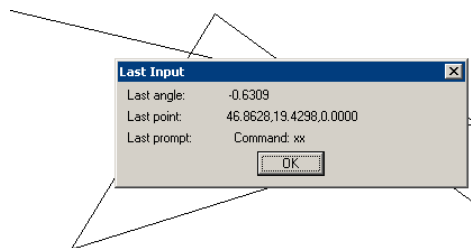
```
: button {  
  key = "okButton";  
  label = "OK";  
  is_default = true;  
  alignment = centered;  
  fixed_width = true;  
}
```

...and then rerun the *xx.lsp* file to see that the centered OK button is properly sized.



Testing the Dialog Box

It's always a good idea to test the dialog box under a number of situations. Use the **Line** command to draw a few lines, and then run the *xx.lsp* routine. The values displayed by the dialog box should be different.



Defining the Command

So far, we've been running *xx.lsp* to develop and test the dialog box. Now that it's running properly, we should change *xx* to something more descriptive. Rename the LISP file as *last.lsp*, and change the function name inside to **C:last**, and make the variables local, as follows:

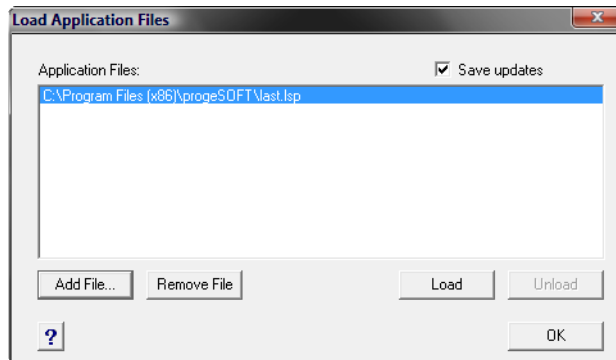
```
(defun c:last (/ dlg-id lang ipt lcmd)
  (setq dlg-id (load_dialog "c:\\lastInput"))
  (new_dialog "lastInput" dlg-id)
  (setq lang (getvar "lastangle"))
    (set_tile "lastAngleData" (rtos lang))
  (setq ipt (getvar "lastpoint"))
    (set_tile "lastPointData" (strcat (rtos (car ipt)) "," (rtos (cadr ipt)) "," (rtos (caddr ipt))))
  (setq lcmd (getvar "lastprompt"))
    (set_tile "lastPromptData" lcmd)
  (action_tile "okButton" "(done_dialog)")
  (start_dialog)
  (unload_dialog dlg-id)
)
```

The DCL file looks like this, in its entirety:

```
lastInput: dialog {
  label = "Last Input";
  : row {
    : text {
      label = "Last angle: ";
      key = "lastAngle";
    }
    : text {
      value = "";
      key = "lastAngleData";
      width = 33;
    }
  }
  : row {
    : text {
      label = "Last point: ";
      key = "lastPoint";
    }
    : text {
      value = "";
      key = "lastPointData";
      width = 33;
    }
  }
  : row {
    : text {
      label = "Last prompt: ";
      key = "lastPrompt";
    }
    : text {
      value = "";
      key = "lastPromptData";
      width = 33;
    }
  }
}
```

```
}  
: button {  
    key = "okButton";  
    label = "OK";  
    is_default = true;  
    alignment = centered;  
    fixed_width = true;  
}  
}
```

If you would like to have this command loaded automatically each time you start progeCAD, add last.lsp to the **AppLoad** command's list of files to load.



Examples of DCL Coding

With the basic tutorial behind you, let's take a look at how to code other types of dialog box features. In this last part of the chapter, we look at how to code the following tiles:

- Buttons
- Check boxes (toggles)
- Radio buttons
- Clusters (columns and rows)

Recall that two pieces of code are always required: (1) the DCL code that specifies the layout of the dialog box, and (2) the LSP code that activates the dialog box.

The next chapter provides you with a comprehensive reference to all DCL tiles, their attributes, and related LISP functions.

Buttons

In the preceding tutorial, you coded an OK button that allowed you to exit the dialog box. It turns out that you don't need to do that, because DCL pre-codes a number of buttons and other dialog box elements for you. These are in a file called *ourbase.dcl* that is normally loaded into progeCAD automatically. (The full list is provided in the next chapter).

The names of the pre-built buttons are:

Prebuilt Tile	Button(s) Displayed
ok_only	OK
ok_cancel	OK Cancel
ok_cancel_help	OK Cancel Help
ok_cancel_help_info	OK Cancel Help Info...
Ok_Cancel_Help_Errtile	OK Cancel Help <i>plus space for error messages.</i>



Use these prebuilt tiles to ensure a consistent look for your dialog boxes. Here is an example of how to use these buttons in DCL files:

```
ok_only;
```

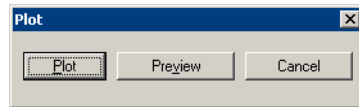
It's that simple. Notice that the tile name lacks the traditional colon (:) prefix, but does require the semi-colon (;) terminator.

DCL allows you to create buttons that have text for labels (**button** tile) or images (**image_button** tile). To indicate that a button opens another dialog box, use an ellipsis (...), as illustrated by the **Info...** button, above.

In addition to text and image buttons, settings can be changed with check boxes (**toggle** tile) and radio buttons (**radio_button** tile), as described next.

Making Buttons Work

It's one thing to populate a dialog box with buttons; it's another to have them execute commands. **OK** and **Cancel** are easy, because their functions have already been defined for you. Let's see how to make buttons labeled **Plot** and **Preview** work. The dialog box looks like this:



The purpose of the Plot button is to execute the **Plot** command, and Preview button the **Preview** command.

(The easy solution would be add an **action** attribute to each button, because it executes LISP functions, such as **command "plot"**. But we cannot use it, because LISP does not allow the highly-useful **command** function to be used in this attribute.)

The key to this problem is the **key** attribute, which gives buttons identifying names by which LISP functions can reference them, such as:

```
key = "plot";
```

Then, over in the LISP file, you use the **action_tile** function to execute the Plot command. Well, not quite. It has the same restriction against use of the **command** function, so we must approach this indirectly by getting **action_tile** to refer to a second LISP routine, such as (**action_tile "plot" "(cmd-plot)"**).

But even this will not work, because we need our dialog box to disappear from the screen, and be replaced by the Plot dialog box. The solution is to get even more indirect:

```
(action_tile "plot" "(setq nextDlg 1) (done_dialog)")
```

"plot" — identifies the Plot button through its key, "plot".

(setq nextDlg 1) — records that the user clicked the Plot button, for further processing later on.

(done_dialog) — closes the dialog box.

This is done twice, once for when the user clicks the Plot button, and again for when the user clicks the Preview button. The Preview button's code is similar; changes are shown in boldface:

```
(action_tile "preview" "(setq nextDlg 2) (done_dialog)")
```

Then we need some code that decides what to do when **nextDlg** is set to 1 or 2:

```
(if (= nextDlg 1) (cmd-plot))  
(if (= nextDlg 2) (cmd-preview))
```

When **nextDlg = 1**, then the following subroutine is run:

```
(defun cmd-plot ()  
  (initia 1)  
  (command "plot")  
)
```

The purpose of the **initdia** function is to force progeCAD to display the dialog box of the Plot command; otherwise, the prompts are displayed at the command line. If that's what you prefer, then comment out the line of code:

```
(defun cmd-plot ()  
  (initdia 1)  
  (command "plot")  
)
```

When nextDlg = 2, then the following subroutine is run:

```
(defun cmd-preview ()  
  (command "preview")  
)
```

Let's look at all the code. First, in the DCL file, we add the key attributes to each button, as follows:

```
x: dialog { label = "Plot";  
  : row {  
    : button { label = "Plot"; mnemonic = "P"; key = plot; }  
    : button { label = "Preview"; mnemonic = "V"; key = "preview"; }  
    cancel_button;  
  } }  
}
```

Second, in the LISP file, we add the code that executes the Plot and Preview commands. Code that relates to the Plot button is shown boldface, while Preview-related code is shown in color:

```
(defun c:xx (/)  
  (setq dlg-id (load_dialog "c:\\x"))  
  (new_dialog "x" dlg-id)  
    (action_tile "plot" "(setq nextDlg 1) (done_dialog)")  
    (action_tile "preview" "(setq nextDlg 2) (done_dialog)")  
  (start_dialog)  
  (unload_dialog dlg-id)  
  (if (= nextDlg 1) (cmd-plot))  
  (if (= nextDlg 2) (cmd-preview))  
)  
  
(defun cmd-plot ()  
  (initdia 1)  
  (command "plot")  
)  
  
(defun cmd-preview ()  
  (command "preview")  
)
```

Check Boxes

Check boxes are created by the **toggle** tile. Check boxes allow you to have one or more options turned on. This is in contrast to radio buttons, which limit you to a single choice.

Let's create a checkbox that changes the shape of point objects through the **PdMode** system variable. (Yes, there is the **DdPType** command, but we'll make ours different.) The system variable can have these values...

PdMode	Meaning
0	Dot (.)
1	Nothing
2	Plus (+)
3	Cross (x)
4	Short vertical line ()
32	Circle
64	Square

... in addition, you can have any combination of these numbers. For example, 34 makes a circle with a plus symbol (32 + 2). A peculiarity: 32 is actually a circle with dot (32 + 0), while 33 is just the circle (32 + 1). Same goes for the square.

Let's create a dialog box that lets us select combinations of plus, circle, and square. To make the dialog box look like this...

```
x: dialog { label = "Point Style";  
  : column { label = "Select a point style: " ;  
    : toggle { key = "plus" ; label = "Plus" ; value = "1" ; }  
    : toggle { key = "circle" ; label = "Circle" ; }  
    : toggle { key = "square" ; label = "Square" ; }  
  }  
  ok_cancel;
```

... takes this code:

```
x: dialog { label = "Point Style";  
  : column { label = "Select a point style: " ;  
    : toggle { key = "plus" ; label = "Plus" ; value = "1" ; }  
    : toggle { key = "circle" ; label = "Circle" ; }  
    : toggle { key = "square" ; label = "Square" ; }  
  }  
  ok_cancel;  
}
```

Notice that **value = "1"** turns on the Plus option (shows a check mark). Now we need to turn to the LSP file and make this dialog box work. Something as simple as (action_tile "plus" "(setvar "pdmode" 2)") doesn't work, because users might select more than one option — which is the whole point to toggles.

We need to (1) read what the users have checked, (2) add up the settings, and then (3) set **PdMode** to show the desired point style.

1. To read user input from dialog boxes, you employ AutoLISP's **\$value** variable, like this for the Plus toggle:

```
(action_tile "plus" "(setq plusVar $value)")
```

Repeat the code for the other two toggles, Circle and Square:

```
(action_tile "circle" "(setq circleVar $value)")
(action_tile "square" (setq squareVar $value))
```

2. The \$value variable contains just 1s and 0s. We will use a lookup table to convert these into the values expected by **PdMode**. For instance, if Plus is selected ("1"), then PdMode expects a value of 2. The lookup table uses the **if** function:

```
(if (= plusVar "1") (setq plusNum 2) (setq plusNum 0))
```

This can be read as:

```
if plusVar = 1, then set plusNum = 2;
otherwise, set plusNum = 0.
```

Repeat the lookup code for the other two toggles, Circle and Square:

```
(if (= squareVar "1") (setq squareNum 64) (setq squareNum 0))
(if (= circleVar "1") (setq circleNum 32) (setq circleNum 0))
```

TIP The \$value retrieved by **get_tile** is actually a string, like "1". The **PdMode** system variable, however, expects an integer. Thus, the lookup table performs the secondary function of "converting" strings to integers.

With the values set to what PdMode expects, add them up:

```
(setq vars (+ plusNum circleNum squareNum))
```

3. To change the value of **PdMode**, you employ AutoLISP's **setvar** function, like this:

```
(setvar "pdmode" vars)
```

Now that we're done, here is all of the LSP code:

```
(defun c:xx (/
  (setq dlg-id (load_dialog "c:\\x"))
  (new_dialog "x" dlg-id)

  ;; Get the current values from each toggle tile:
  (setq plusVar (get_tile "plus"))
  (setq circleVar (get_tile "circle"))
  (setq squareVar (get_tile "square"))

  ;; See which toggles the user clicks:
  (action_tile "plus" "(setq plusVar $value)")
  (action_tile "circle" "(setq circleVar $value)")
  (action_tile "square" "(setq squareVar $value)")

  (start_dialog)
  (unload_dialog dlg-id)

  ;; Lookup table converts "0"/"1" strings to the correct integers:
  (if (= plusVar "1") (setq plusNum 2) (setq plusNum 0))
  (if (= circleVar "1") (setq circleNum 32) (setq circleNum 0))
  (if (= squareVar "1") (setq squareNum 64) (setq squareNum 0))
```

```

;; Add up the integers, and then change system variable:
(setq vars (+ plusNum circleNum squareNum))
(setvar "pdmode" vars)
)

```

Here are some of the point styles generated by this dialog box:



Radio Buttons

Radio buttons are easier than toggles, because only one can be active at a time. Let's create a dialog box that uses radio buttons to set the isoplane. The dialog box changes the value of the **SnapIsoPair** system variable:

SnapIsoPair	Meaning
0	Left isoplane (default).
1	Top isoplane.
2	Right isoplane.

To make the dialog box look like this...

```

x: dialog { label = "Isoplane";
: column { label = "Change the isoplane to: ";
: radio_button { key = "left" ; label = "Left isoplane" ; value = "1" ; }
: radio_button { key = "top" ; label = "Top isoplane" ; }
: radio_button { key = "right" ; label = "Right isoplane" ; }
spacer;
ok_cancel;

```

... takes this code:

```

x: dialog { label = "Isoplane";
: column { label = "Change the isoplane to: ";
: radio_button { key = "left" ; label = "Left isoplane" ; value = "1" ; }
: radio_button { key = "top" ; label = "Top isoplane" ; }
: radio_button { key = "right" ; label = "Right isoplane" ; }
spacer;
}
ok_cancel;
}

```

Notice that **value = "1"** turns on the Left isoplane option (shows a dot).

Before going on to the accompanying LSP file, set up progeCAD to display isometric mode:

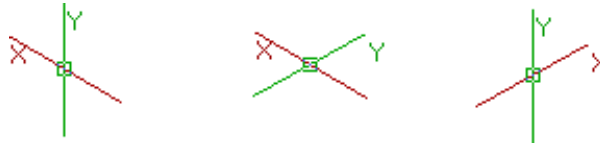
1. Enter the **Settings** command.

If necessary, choose the **Coordinate Input** tab, and then select Snap and Grid from the Change Settings For drop list.

2. Select **Isometric snap and grid**.

3. Click **OK**.

progeCAD is now in isometric mode. As you use the dialog box described below, the cursor changes its orientation:



Left: Cursor for the left isoplane.

Center: Cursor for the top isoplane.

Right: Cursor for the right isoplane.

Let's now turn to the LISP file to make this dialog box work. It is similar to the code used for toggles; the primary difference is that the values are not added up:

```
(defun c:xx (/
  (setq dlg-id (load_dialog "c:\\x"))
  (new_dialog "x" dlg-id)

  ;; See which radio button the user clicks:
  (action_tile "left" "(setq leftVar $value)")
  (action_tile "top" "(setq topVar $value)")
  (action_tile "right" "(setq rightVar $value)")

  (start_dialog)
  (unload_dialog dlg-id)

  ;; Lookup table:
  (if (= leftVar "1") (setq vars 0))
  (if (= topVar "1") (setq vars 1))
  (if (= rightVar "1") (setq vars 2))

  ;; Change system variable:
  (setvar "snapisopair" vars)
)
```

We have been cheating, because we forced the dialog box to show the Left isoplane as the default. This is not necessarily true. We really should modify the DCL and LISP code to make the dialog box show which isoplane is the default. This is done with AutoLISP's **set_tile** function.

First, change the DCL code so that it no longer makes the Left isoplane the default: change value = "1" to "":

```
value = ""
```

In the LSP code, we need to (1) extract the value of SnapIsoPair with **getvar**, and then (2) use **set_tile** as a callback.

1. Extract the current value of **SnapIsoPair**:

```
(setq vars (getvar "snapisopair"))
```

2. Set the default button:

```
(if (= vars 0) (set_tile "left" "1"))
```

This reads, as follows:

If the value of SnapIsoPair is 0 (= vars 0),
then turn on the Left isoplane radio button (set_tile "left" "1").

Do similar code for the other two radio buttons:

```
(if (= vars 1) (set_tile "top" "1"))  
(if (= vars 2) (set_tile "right" "1"))
```

The other change we need to make is to change some of the variables to local:

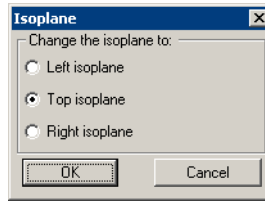
```
(defun c:xx (/ leftVar topVar rightVar)
```

This forces the three variables to lose their values when the LSP routine ends. The reason they need to do this is because otherwise **rightVar** keeps its value (it's the last one) and makes Right isoplane the default every time.

With these three changes in place, the improved code looks like this — with changes highlighted in boldface:

```
(defun c:xx (/ leftVar topVar rightVar)  
  (setq vars (getvar "snapisopair"))  
  (setq dlg-id (load_dialog "c:\\x"))  
  (new_dialog "x" dlg-id)  
  
  :: Set the default button:  
  (if (= vars 0) (set_tile "left" "1"))  
  (if (= vars 1) (set_tile "top" "1"))  
  (if (= vars 2) (set_tile "right" "1"))  
  
  ;; See which radio button the user clicks:  
  (action_tile "left" "(setq leftVar $value)")  
  (action_tile "top" "(setq topVar $value)")  
  (action_tile "right" "(setq rightVar $value)")  
  
  (start_dialog)  
  (unload_dialog dlg-id)  
  
  ;; Lookup table:  
  (if (= leftVar "1") (setq vars 0))  
  (if (= topVar "1") (setq vars 1))  
  (if (= rightVar "1") (setq vars 2))  
  
  ;; Change system variable:  
  (setvar "snapisopair" vars)  
)
```

Now each time the dialog box starts, it correctly displays the default isoplane, such as “top” as illustrated below:



Clusters

Clusters help you combine related groups of controls. DCL lets you have vertical, horizontal, boxed, and unboxed clusters. Radio clusters are required when you want to have two radio buttons on at the same time. Otherwise, clusters are needed only for visual and organizational purposes.

DCL makes it look as if there are eight tiles for creating clusters:

Column	Row
Boxed_Column	Boxed_Row
Radio_Column	Radio_Row
Boxed_Radio_Column	Boxed_Radio_Row

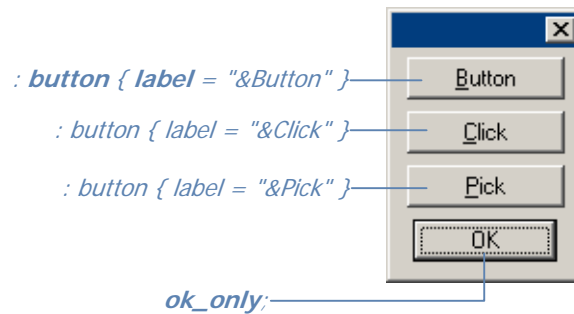
These eight options can be reduced to three when you take the following into account:

- The **column** tile is usually not needed, because progeCAD automatically stacks tiles vertically into a column.
- The **column** and **row** tiles display a box as soon as you include a label for them.
- Tiles with **radio** in their names are only for clustering radio buttons.

Columns and Rows

progeCAD normally stacks tiles, so no **column** tile is needed, as illustrated by this DCL code:

```
x: dialog {
  : button { label = "&Button" }
  : button { label = "&Click" }
  : button { label = "&Pick" }
  ok_only;
}
```



(The ampersand — **&** — specifies the shortcut keystroke that accesses the button from the keyboard with the **Alt** key, such as pressing **Alt+B**.) To create a horizontal row of tiles, use the **row** tile, as shown in boldface below:

```
x: dialog {  
    : row {  
        : button { label = "&Button" }  
        : button { label = "&Click" }  
        : button { label = "&Pick" }  
    }  
    ok_only;  
}
```

The horizontal row is invisible, so I highlighted it with the white rectangle in the figure below.



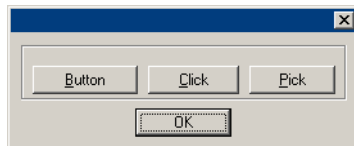
Because the **ok_only** tile is outside of the **row {}** tile, it is located outside of the cluster, stacked vertically below the row of three buttons.

Boxed Row

To display a rectangle (a.k.a. box) around the buttons, use the **boxed_row** tile, as follows:

```
x: dialog {  
    : boxed_row {  
        // et cetera  
    }  
    ok_only;  
}
```

The box is made of white and gray lines to give it a chiseled 3D look.

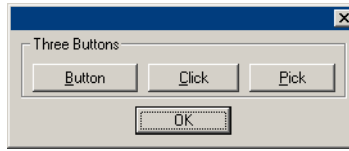


Boxed Row with Label

You can add text to describe the purpose of the boxed buttons with the **label** attribute, as shown in boldface below:

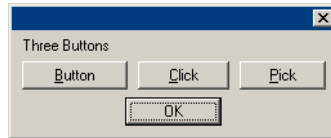
```
x: dialog {  
    : boxed_row { label = "Three Buttons";  
        // et cetera  
    }  
    ok_only;  
}
```

The curious thing is that you get the same effect whether using the **boxed_row** or **row** tile. That's right: when you add a label to the **row** tile, progeCAD displays a box around the cluster.



To avoid the box, precede the row with the **text** tile for the title, as follows:

```
x: dialog {  
  : text { label = "Three Buttons";}  
  : row {  
    // et cetera  
  }  
  ok_only;  
}
```



Special Tiles for Radio Buttons

You can use the regular row and column tiles with radio buttons, except in one case: when you need more than one radio button to be turned on. Recall that only one radio button can be on (show the black dot) at a time; progeCAD automatically turns off all other radio buttons that might be set to on (**value** = "1").

The solution is to use two or more **radio_column** tiles, each holding one of the radio button sets that need to be on. Examples of this are given in the next chapters.

It is not recommended to use rows for radio buttons, because they make it psychologically more difficult for users to choose.

Debugging DCL

The most common errors are due to errors in punctuation, such as leaving out a closing semi-colon or quotation mark. These problems are announced by dialog boxes illustrated later in this section.

Dcl_Settings

DCL contains a debugger. Add the following code to the beginning of the DCL file, before the dialog tile:

```
dcl_settings : defalut_dcl_settings { audit_level = 3 ; }
x : dialog { // et cetera
```

The debugger operates at four levels:

Audit Level	Meaning
0	No debugging performed.
1	(Default) Checks for DCL errors that may terminate progeCAD,s uch as undefined tiles and circular prototype definitions.
2	Checks for undesirable layouts and behaviors such as missing attributes and wrong attribute values.
3	Checks for redundant attribute definitions.

DCL Error Messages

progeCAD displays DCL-related error messages in dialog boxes. Some of the ones you may encounter include these:

Dialog has neither an OK nor a CANCEL button

Dialog boxes need to exit through the **OK** or **Cancel** button. At the very least, add the **ok_only** tile to the DCL file. (DCL was written before Windows automatically added the **x** (cancel) button to all dialog boxes, and Autodesk has failed to update DCL to take this innovation into account.)

Error in dialog file "filename.dcl", line *n*

Your DCL file contains the name of a tile unknown to progeCAD. Check the spelling. In this specific example, **ok_only** was prefixed by a colon (:), which is incorrect for prebuilt tiles.

```
Incorrect:      : ok_only ;
Correct:       ok_only ;
```

Dialog too large to fit on screen

A tile in the DCL file is creating a dialog box that would not fit your computer's screen. This can happen when the **edit_edith**, **width**, or **height** attributes are too large.

Additional Resources

In addition to the information provided by this ebook, you may wish to refer to other DCL references. These include:

- AfraLisp at www.afralisp.net/lisp/dialog1.htm has lots of tutorials on programming progeCAD, including these DCL topics:

- DCL Primer - Download
- Dialog Box Layout
- Dialogue Boxes Step by Step
- Dialogue Boxes in Action
- Nesting and Hiding Dialogues
- Hiding Dialogues Revisited
- AutoLisp Message Box
- AutoLisp Input Box
- Referencing DCL Files
- AutoLISP Functions for Dialog Control Language
- Functional Synopsis of DCL Files
- DCL Attributes
- Dialogue Box Default Data
- DCL Model
- DCL Progress Bar
- Attributes and Dialogue Boxes
- DCL without the DCL File
- Entering Edit Boxes

- Jeffery Sanders has his “The AutoLisp Tutorial - DCL: Dialog Control Language” tutorials at www.jefferysanders.com/autolisp_DCL.html. Topics include:

- Part 1 - Buttons
- Part 2 - Edit_Box
- Part 3 - List_Box
- Part 4 - PopUp_List
- Part 5 - Radio_Buttons
- Part 6 - Text and Toggle
- Part 7 - Putting it all together

- The AutoLISP Exchange presents “Getting Started with Dcl Dialogs” tutorials and several DCL utility programs at web2.airmail.net/terrycad/Tutorials/MyDialogs.htm. The tutorial topics include:

- Tutorial Overview
- Download Files
- Introduction to AutoLISP
- Introduction to Dialogs
- My Dialogs Menu
- Questions & Comments

The utility programs are:

Dcl Calcs	View Dcl
Show Icons	Get Icon
Get Buttons	

Additional utilities are listed at web2.airmail.net/terrycad/, such as *GetVectors* for creating images for dialog boxes from CAD entities.

DCL Reference

dCL allows you to create these elements in dialog boxes: buttons, popup lists, text edit boxes, radio buttons, image buttons, sliders, list boxes, and toggles.

These elements are called *tiles*, and can be clustered together as dialog boxes, boxed columns, boxed radio columns, radio columns, boxed radio rows, radio rows, boxed rows, rows, and columns. To make dialog boxes prettier and show graphical information, you can add these elements: images, spacer 0, text, spacer 1, and spacer.

The *base.dcl* file defines numerous basic tiles, such as the OK button, so that you don't need to write them from scratch. This file is reproduced in full later in this chapter.

Each tile works with one or more attributes. Attributes specify the look of the tile and how it works. For instance, the label tile specifies the text that appears on buttons. A special attribute, called "key," allows LISP code to communicate back to the dialog box and make changes, such as changing the text displayed by the dialog box's title bar.

This chapter describes every tile and its associated attributes, as well as the LISP functions that are specific to dialog boxes.

TIP DCL is under development in progeCAD and all functions described in this chapter do not necessarily operate as may be expected.

In This Chapter

- Alphabetical summary of DCL tiles
- Alphabetical summary of DCL attributes
- Tile reference
- Ourbase.dcl

Alphabetical Summary of DCL Tiles

<code>boxed_column</code>	Draws a rectangle around a vertical column of tiles.
<code>boxed_radio_column</code>	Draws a rectangle around a vertical column of radio tiles.
<code>boxed_radio_row</code>	Draws a rectangle around a horizontal row of radio tiles.
<code>boxed_row</code>	Draws a rectangle around a horizontal row of tiles.
<code>button</code>	Displays a button with text.
<code>column</code>	Creates a column of tiles.
<code>dialog</code>	Creates a dialog box.
<code>default_dcl_settings</code>	Sets the level of debugging.
<code>edit_box</code>	Displays a text edit box.
<code>image</code>	Displays a static image.
<code>image_button</code>	Displays a button with an image.
<code>list_box</code>	Displays a list.
<code>paragraph</code>	Concatenates text tiles into vertical paragraphs.
<code>popup_list</code>	Displays a droplist.
<code>radio_button</code>	Displays a round radio button.
<code>radio_column</code>	Creates a column of radio buttons.
<code>radio_row</code>	Creates a row of radio buttons.
<code>row</code>	Creates a row of tiles.
<code>slider</code>	Displays a vertical or horizontal slider bar.
<code>spacer</code>	Inserts a rectangular space.
<code>spacer_0</code>	Inserts variable-width space.
<code>spacer_1</code>	Inserts narrow space.
<code>text</code>	Displays static text.
<code>text_part</code>	Contains piece of text.
<code>toggle</code>	Displays a square checkbox.

Alphabetical Summary of DCL Attributes

action	LISP action expression.
alignment	Horizontal or vertical position in a cluster.
allow_accept	Activates the is_default attribute when tile is selected.
aspect_ratio	Aspect ratio of an image.
audit_level	Specifies the level of debugging.
big_increment	Incremental distance to move.
children_alignment	Alignment of a cluster's children.
children_fixed_height	Height of a cluster's children doesn't grow during layout.
children_fixed_width	Width of a cluster's children doesn't grow during layout.
color	Background (fill) color of an image.
edit_limit	Maximum number of characters that can be entered.
edit_width	Width of the input field of the tile.
fixed_height	Prevents height from shrinking.
fixed_width	Prevents width from shrinking.
fixed_width_font	Displays text in a fixed pitch font.
height	Height of the tile.
initial_focus	Key of the tile with initial focus.
is_bold	Displays as bold.
is_cancel	Reacts to the cancel key (Esc).
is_default	Reacts to the accept key (Enter).
is_enabled	Tile is initially enabled.
is_tab_stop	Tile is a tab stop.
key	Tile name used by the application.
label	Displayed label of the tile.
layout	Whether the slider is horizontal or vertical.
list	Initial values to display in list.
max_value	Maximum value of a slider.
min_value	Minimum value of a slider.
mnemonic	Mnemonic character for the tile.
multiple_select	List box allows multiple items to be selected.
password_char	Masks characters entered in edit_box.
small_increment	Incremental distance to move.
tab_truncate	Truncates text longer larger than the associated tab stop.
tabs	Tab stops for list display.
value	Tile's initial value.
width	Width of the tile.

Tile Reference

This reference lists DCL's tiles and attributes in order of importance, as follows:

- Dialog
- Button
 - Ok_Only
 - Ok_Cancel
 - Ok_Cancel_Help
 - Ok_Cancel_Help_Errtile
 - Ok_Cancel_Help_Info
- Radio_Button
- Toggle
- Image_Button
- Edit_Box
- List_Box
- Popup_list
- Slider
- Text
 - Text_Part
 - Concatenation
 - Paragraph
 - Errtile
- Spacer
 - Spacer_0
 - Spacer_1
 - Image
- Column
- Row
 - Boxed_Column
 - Boxed_Row
 - Radio_Column
 - Radio_Row
 - Boxed_Radio_Column
 - Boxed_Radio_Row

The default value of attributes is shown in **boldface**. For example,

alignment = **left** | right | centered;

means that “left” is the default for the Alignment attribute.

Dialog

The **dialog** tile defines dialog boxes.



```
name : dialog {  
  label = "text";  
  value = "text";  
  initial_focus = "key";  
}
```

Name

The **name** attribute identifies dialog boxes by name. This allows you to have all dialog boxes in a single DCL file. The LISP routine that accompanies the DCL file uses the **load_dialog** function to load the *filename.dcl* file, and then uses **new_dialog** to locate the specific dialog box, as follows:

```
(setq dlg-id (load_dialog "c:\\filename"))  
(new_dialog "name" dlg-id)
```

TIP The *dlg-id* variable holds the system-assigned identifier for the DCL file. This is typically a number, such as 30.
If the number has a negative value, such as -1, then the DCL file failed to load correctly. You can use this number to generate error reports.

Label

The **label** attribute displays text on the dialog box's title bar, such as:

```
label = "Dialog Box";
```

The **value** attribute is nearly the same, because it also displays text on the title bar. The difference is that you can use the LISP **set_value** function to later change the title.

TIP You can change the dialog box's title when the accompanying LISP routine is run. This is useful, for example, with a file dialog box whose title should reflect the extension of the file extension being accessed.
To change the text, use the (**set_tile** *key* *value*) function, which changes the value of the tile specified by *key*.
The problem is that progeCAD cannot widen the dialog box to accommodate a longer title. To avoid cutting off some of the **value** text, specify a long title with **label**.

Initial Focus

The **initial_focus** attribute indicates which button or other tile gets the *focus*. "Focus" refers to the tile that is highlighted, the one that would be activated when you press the **Enter** key.



Usually, the focus is set to the OK button, or to the tile users are likely to access the most.


Key

The value of `focus` is the name of the **key** tied to the tile. For example, when the key of the OK button is “okButton,” then you enter the following:

```
initial_focus = "okButton";
```

Exiting Dialog Boxes

Every dialog box must have at least an **OK** button, so that users can exit the dialog box. You can use predefined buttons to give your dialog boxes the same look as those of progeCAD-designed dialog boxes. These are called “subassemblies,” and are found in *base.dcl*.

For instance, to include the standardized  button, use the **ok_only** subassembly like this:

```
name : dialog {  
    label = "Dialog Box";  
    ok_only;  
}
```

Notice that subassemblies are not prefixed by the `:` character. (If progeCAD complains about the `ok_only` subassembly, then you need to load the *base.dcl* file with the following bit of code:

```
Command: (load_dialog "base.dcl")
```

Both OK and Cancel exit the dialog box, but they have different meanings:

- **OK** records changes made by users.
- **Cancel** discards the changes.

After the dialog box is exited, progeCAD sets the read-only **DiaStat** system variable (short for “dailog box status”) to one of the following values:

DiaStat	Meaning
0	User clicked Cancel to exit dialog box.
1	User clicked OK to exit dialog box.

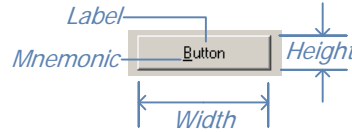


At the right end of the dialog box’s title bar is an **x** button. It is equivalent to **Cancel**, and users can use it in place of Cancel. However, DCL was developed before Windows added the x button to all dialog boxes, so progeCAD does not recognize that leaving out Cancel is now valid when the OK button is also missing.

If the dialog box is tight on space, you can leave out the Cancel button, and let users use the x button; just remember to include the OK button.

Button

The **button** tile defines buttons with text labels.



```
: button {  
  label = "text";  
  mnemonic = "char";  
  
  action = "(LISP function)";  
  key = "text";  
  
  is_cancel = false | true;  
  is_default = false | true;  
  is_enabled = true | false;  
  is_tab_stop = true | false;  
  
  width = number;  
  height = number;  
  fixed_height = false | true;  
  fixed_width = false | true;  
  
  alignment = centered | left | right;  
}
```

Label

The **label** attribute places text on the button tile, such as:

```
label = "Button";
```

Mnemonic

The **mnemonic** attribute underlines a character. Users can then access the button by pressing **Alt** and the letter. For example, the following code underlines the letter “B” in Button:

```
mnemonic = "B";
```

TIPS As an alternative to the mnemonic attribute, you can prefix the character with **&** in the label attribute, like this:
label = "**&**Button";

Make sure that mnemonic characters are not used more than once in each dialog box. For instance, don't use B twice in the same dialog box.

Action

The **action** attribute contains LISP code that gets executed when users click the button. (This is called a “callback.”) For example, the code could set the value of system variables, like this:

```
action = "(setvar "highlight" 0)"
```

TIPS You cannot, unfortunately, use the LISP **command** function to execute progeCAD commands with the **action** attribute.

You can use the LISP **action_tile** function to override the action specified by the **action** attribute.

Key

The **key** attribute gives an identifying tag to the button.

Is_Cancel

The **is_cancel=true** attribute specifies that this button is selected when users press the **Esc** key.

```
is_cancel = true;
```

Usually, the dialog box is exited right away when users press **Esc**. In addition, progeCAD sets the value of the **DiaStat** system variable to **0**. However, if the button has an **action** attribute, then the associated LISP expression is executed before the dialog box is exited.

TIPS Only one button in the dialog box can be assigned **is_cancel=true**.

There is no point in having **is_cancel=false**, except for debugging perhaps.

Is_Default

The **is_default=true** attribute specifies that this button is selected when users press the **Enter** key — unless another button has the focus.

```
is_default true;
```

Is_Enable

The **is_enable=false** attribute allows you to gray-out buttons. This tells users that the buttons are unavailable, usually because some other condition is not satisfied, such as the drawing is in paper space instead of model space.

```
is_enabled= false;
```

Grayed-out button — 

When set to true, the buttons become available. To change the status from false to true, use the **mode_tile** function in LISP.

Is_Tab_Stop

The **is_tab_stop** attribute allows the button to receive focus when users press the **Tab** key. Pressing Tab is a popular way for power users to quickly move through the controls of dialog boxes; if the mouse is busted, then that's the only way to navigate a dialog box. Normally, there is no reason **not** to allow a button to be a tab stop, and since the default is true, there's not much need for this attribute.

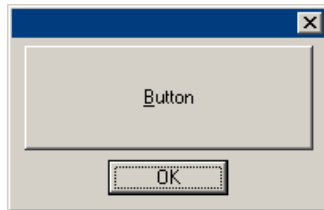
```
is_tab_stop= false;
```

Width & Height

The **width** and **height** attributes specify the minimum size of buttons. You can use integers (such as 5) or real numbers (such as 5.5).

Usually progeCAD determines the correct size on its own, so you don't need to specify these attributes. But if you need to create extra large buttons, such as the one illustrated below, then go right ahead!

```
width = 30;  
height = 5;
```



The units are in characters, such as 30 characters wide and 5 lines tall. progeCAD determines the size of character based on an average calculated of all letters in the 8-pt "MS San Serif" font used for text in dialog boxes. The font cannot be changed.

In the table below, the black areas indicate size of tiles based on a variety of values for the **Width** and **Height** attributes:

Size	Example
Height = 1	
Height = 2	
Height = 3	
<hr/>	
Width = 30	
Width = 40	
Width = 80	

Fixed_Height & Fixed_Width

The **fixed_height** and **fixed_width** attributes prevent progeCAD from expanding buttons to fill the available space. Recall that the **height** and **width** attributes specify only the minimum size. Adding these two attributes makes them also the maximum size.

```
fixed_height = true;  
fixed_width = true;
```

TIP Use the **image_button** tile for buttons with colors and images.

Alignment

The **alignment** attribute is supposed to shift text left or right on the button. In practice, however, I find this attribute has no effect; the text is always centered.

Autodesk notes that alignment cannot be specified along the long axis of a cluster of tiles. The first and last tiles align with either end of the row (or column), while the inner tiles are distributed evenly between them. You can change the distribution with the **spacer_0** tile.

Prefabricated Button Assemblies

progeCAD provides the following pre-fabricated button assemblies in *ourbase.dcl*. This file is described in detail later in this chapter.

Ok_Only

The **ok_only** tile defines an OK button.

ok_only;



Ok_Cancel

The **ok_cancel** tile defines a horizontal row of OK and Cancel buttons.

ok_cancel;

Ok_Cancel_Help

The **ok_cancel_help** tile defines a horizontal row of OK, Cancel, and Help buttons.

ok_cancel_help;



Ok_Cancel_Help_Errtile

The **ok_cancel_help_errtile** tile defines a horizontal row of OK, Cancel, and Help buttons, and space below for an error message.

ok_cancel_help_errtile;

Ok_Cancel_Help_Info

The **ok_cancel_help_info** tile defines a horizontal row of OK, Cancel, Help, and Info buttons. The **Info** button can be used to display a second dialog box with additional information.

ok_cancel_help_info;



Radio_Button

The **radio_button** tile defines radio buttons. These buttons are used when only one choice can be made from a selection, such as the top, left, or right isoplane. When selected, the radio button shows a black dot; when off, the round button is blank.

If the dialog box has more than one radio button in a cluster, only one can be on at a time. When users select a radio button, the other one turns off automatically. To have more than one radio button on at a time, segregate them into clusters with the **radio_row** or **radio_column** tiles.



```
: radio_button {  
  action = "(LISP function)";  
  key = "text";  
  
  label = "text";  
  mnemonic = "char";  
  value = "0" | "1";  
  
  is_enabled= true | false;  
  is_tab_stop= true | false;  
  
  height = number;  
  width = number;  
  fixed_height = false | true;  
  fixed_width = false | true;  
  
  alignment = left | right | centered;  
}
```

Label

The **label** attribute describes the purpose of the radio button to users. The text is always to the right of the button.

Value

The **value** attribute determines whether the radio button is initially on or off:

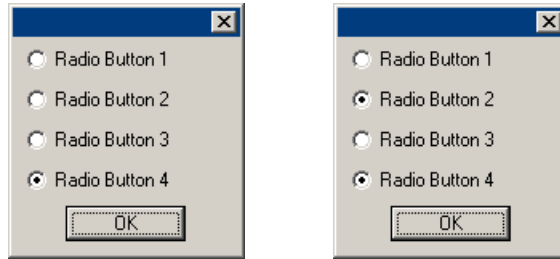
```
value = "1";
```

Value	Meaning	Example
0	Off	<input type="radio"/> Radio button
1	On	<input checked="" type="radio"/> Radio Button

The other attributes have the same meaning as for the **button** tile.

Multiple Radio Buttons

When more than one radio button has **value** set to **1**, then progeCAD turns on only the last one, as illustrated below.



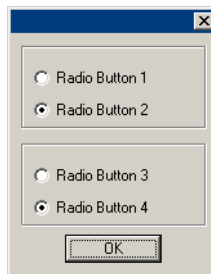
Left: Four radio buttons in one dialog box.

Right: Four radio buttons segregated into two vertical columns.

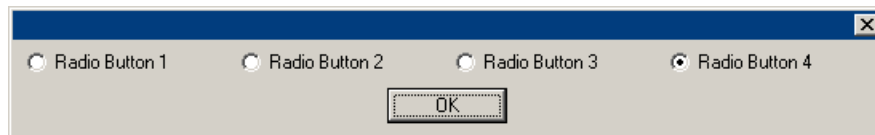
If you need more than one radio button to be turned on, then consider using the **toggle** tile for check boxes. Alternatively, segregate radio buttons with the **radio_column** tile, as illustrated above, and shown in DCL code below:

```
:radio_column {
  :radio_button { label = "Radio Button 1"; value = "1"; }
  :radio_button { label = "Radio Button 2"; value = "1"; }
}
:radio_column {
  :radio_button { label = "Radio Button 3"; value = "1"; }
  :radio_button { label = "Radio Button 4"; value = "1"; }
}
```

It's not clear that the four buttons are segregated into two sections, so it makes more sense to use the **boxed_radio_column** tile, which separates them visually.



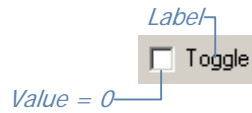
By default, progeCAD stacks radio buttons vertically, as illustrated above. Use the **radio_row** tile to force the radio buttons in a horizontal line — although this format is more difficult for users to navigate.



More on **boxed_** and **radio_** tiles later in this chapter.

Toggle

The **toggle** tile defines check boxes. Check boxes are employed so users can select more than one choice. (Use radio buttons to limit options to a single choice.)



```
: toggle {
  action = "(LISP function)";
  key = "text";

  label = "text";
  mnemonic = "char";
  value = "0" | "1";

  is_enabled= true | false;
  is_tab_stop= true | false;

  height = number;
  width = number;
  fixed_height = false | true;
  fixed_width = false | true;

  alignment = left | right | centered;
}
```

Label

The **label** attribute describes to users the purpose of the check box. The text is always to the right of the button.

Value

The **value** attribute determines whether the toggle is initially on or off:

```
value = "1";
```

Value	Meaning	Example
0	Off	<input type="checkbox"/> Check box
1	On	<input checked="" type="checkbox"/> Check box

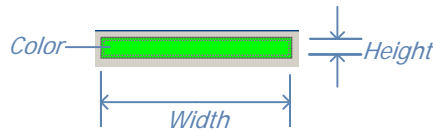
Other Attributes

The other attributes have the same meaning as for the **radio** tile.

You can use the **boxed_row** and **boxed_column** tiles to segregate toggles into groups.

Image_Button

The **image_button** tile defines a button tile with an image. This can be the most difficult tile to program, because some situations require you to correlate x,y coordinates from users' picks with LISP code.



```
: image_button {  
  action = "(LISP function)";  
  key = "text";  
  
  aspect_ratio = number;  
  mnemonic = "char";  
  color = colornumber;  
  
  allow_accept = false | true;  
  is_enabled = true | false;  
  is_tab_stop = true | false;  
  
  height = number;  
  width = number;  
  fixed_height = false | true;  
  fixed_width = false | true;  
  
  alignment = left | right | centered;  
}
```

Key

The **key** attribute identifies the **image** tile to the accompanying LISP code, so that the slide image can be placed in the dialog box.

```
key = "image1";
```

Images of hatch patterns, fonts, and so on are placed on the image tile through the accompanying LISP code's callback function (**set_tile**) and the **key** attribute. There are two sources of image you can use:

- SLD slide files, which are created ahead of time with progeCAD's **MSlide** command, and then placed with LISP's **slide_image** function.
- Vector lines, which are drawn on-the-fly by LISP's **vector_image** function.

Aspect_Ratio, Height, & Width

You use any two of these three attributes. The **aspect_ratio** attribute specifies the ratio between the height and width of the image tile, and must be used with either the **height** or the **width** attribute — but not both. Similarly, if you use the **height** and **width** attributes, you cannot use the **aspect_ratio** attribute. Examples:

```
aspect_ratio = 1.333;  
height = 3;
```

Or...

```
aspect_ratio = 1.333;  
width = 4;
```

Or...

```
height = 3;  
width = 4;
```

Color

The **color** attributes specifies the background color of image tiles. You can use a color name or number; default = 7 (white or black).

mber	Color Name	Meaning
0	black	ACI color 0 (black or white) ¹
1	red	ACI ² color 1
2	yellow	ACI color 2
3	green	ACI color 3
4	cyan	ACI color 4
5	blue	ACI color 5
6	magenta	ACI color 6
7	white	ACI color 7 (white or black) ¹

-1 graphics_foreground Current default color of entities (usually ACI 7). ¹

-2 graphics_background Current background color of progeCAD's graphics screen.

-3 blue

-4 black

-5 gray

-6 black

-7 red

-15 dialog_background Current color of dialog box background (usually gray).

-16 dialog_foreground Current color of dialog box text (usually black).

-18 dialog_line Current color of dialog box lines (usually black).

Notes:

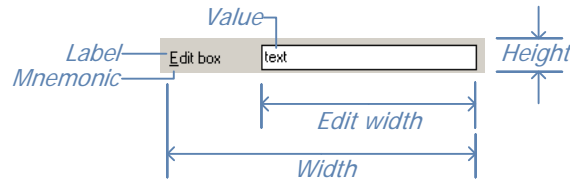
¹ The color is white when the background color is dark, but black when the background is light.

² ACI is short for "AutoCAD Color Index," and refers to the 256 color numbers.

Autodesk notes that "if your image tile is blank when you first display it, try changing its color to graphics_background or graphics_foreground."

Edit_Box

The **edit_box** tile defines a horizontal tile for entering text.



```
: edit_box {  
    label = "text";  
    mnemonic = "char";  
  
    action = "(LISP function)";  
    key = "text";  
  
    value = "text";  
    fixed_width_font = false | true;  
    password_char = "char";  
    edit_limit = 1-256;  
    edit_width = 1-256;  
  
    allow_accept = false | true;  
    is_enabled = true | false;  
    is_tab_stop = true | false;  
  
    height = number;  
    width = number;  
    fixed_height = false | true;  
    fixed_width = false | true;  
  
    alignment = left | right | centered;  
}
```

Label

The **label** attribute displays text that prompts users as to the text or numbers to enter. The label is always positioned to the left of the text entry box.

```
label = "Edit Box";
```

Mnemonic

The **mnemonic** attribute provides the Alt-shortcut keystroke for the label. Alternatively, prefix a letter in the label with **&**.

```
label = "&Edit Box";  
mnemonic = "E";
```

Value

The **value** attribute displays default text in the edit box, such as "text" in the figure above. For blank, leave out value, or set use value = "".

```
value = "text";
```

Password_Char

When the edit box is used for entering passwords, then you can specify a character with the **password_char** attribute that substitutes for user-entered text, such as “*”.

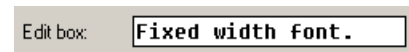
```
password_char = "*";
```



Fixed_Width_Font

The **fixed_width_font** attribute determines whether the edit box uses a fixed width font; more precisely, the monospaced FixedSys font included with Windows. Only the user text is affected by this attribute; the dialog box text keeps its font.

```
fixed_width_font = true;
```



Edit_Limit

The **edit_limit** attribute limits the maximum number of characters users can type in. For text, the limit usually doesn't matter; the default is 132. You might want to expand the limit to its maximum of 256, or reduce it. For example, you may want to limit entry to a single character or digit.

```
edit_limit = 256;
```

Edit_Width

The **edit_width** attribute determines the size of the edit box; it can be an integer or a real number. Users can enter more characters than this number, up to the maximum determined by **edit_limit**. The default width is whatever fits in the dialog box; specifying **edit_width = 0** has the same effect. In many cases, the default width is about 16 characters.

```
edit_width = 186;
```

I have found that the maximum value of 256 overwhelms progeCAD; it complains that the resulting dialog box would not fit my computer's 1280x1024-resolution screen. The maximum turned out to be 186 in my case.

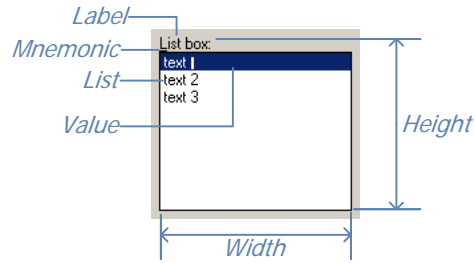


Other Attributes

The remaining attributes have the same meaning as for other tiles.

List_Box

The **list_box** tile defines tiles that list text items; users can select one or more of them.



```
: list_box {
    action = "(LISP function)";
    key = "text";

    label = "text";
    mnemonic = "char";

    list = "text ^\ntext 2^\ntext 3";
    value = "O";
    multiple_select = true | false;
    tabs = "number number number";
    tab_truncate = false | true;
    fixed_width_font = false | true;

    allow_accept = false | true;
    is_enabled = true | false;
    is_tab_stop = true | false;

    height = number;
    width = number;
    fixed_height = false | true;
    fixed_width = false | true;

    alignment = left | right | centered;
}
```

List

The **list** attribute specifies the text in the list box tile. Each item is separated by the `\n` meta-character, which means “new line.” When the list becomes too long for the list box, progeCAD automatically adds a scroll bar, as illustrated later.

```
list = "text ^\ntext 2^\ntext 3";
```

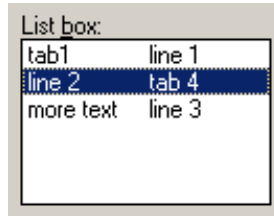
Tabs

You can use the **tabs** attribute to line up text in list boxes. The tabs are specified in characters, such as at the 5th, 10th, 15th, and 20th character.

```
tabs = "5 10 15 20";
```

To specify tabs in the text of the **list** attribute, use the `\t` metacharacter (short for “tab”). The following DCL code and figure illustrate the use of `\n` and `\t`:

```
list = " tab1\nline 1\nline 2\ttab 4\nmore text\nline 3";
```



Tab_Truncate

The **tab_truncate** attribute determines whether text is truncated when longer than the associated tab stop. Default is false, which means it is not truncated.

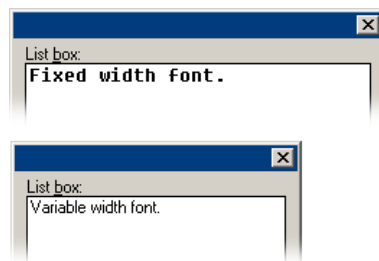
```
tab_truncate = true;
```

Fixed_Width_Font

The **fixed_width_font** attribute lets the list use the Windows FixedSys font, a monospace font (a.k.a. fixed width font), where each character takes up the same width. This can be useful when you need columns of text to line up; otherwise, fixed width text is not useful, because it makes the dialog box wider.

```
fixed_width_font = true;
```

In the figure below, both dialog boxes have **width = 30**. Notice that the fixed width font takes up more space.



Value

The **value** attribute specifies which item is initially highlighted. The default, **0**, means the first item is highlighted. If you want more than one item highlighted, then separate the digits with spaces. The following examples highlights items #2 and #3:

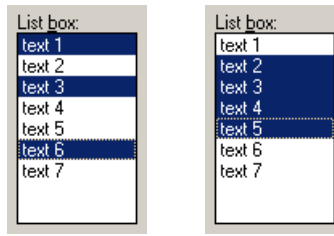
```
value = "1 2";
```

Multiple_Select

The **multiple_select** attribute determines whether users can select more than one item from the list. Users need to hold down the **Ctrl** key to select more than one item, or the **Shift** key to select a sequential range of items.

```
multiple_select = false;
```

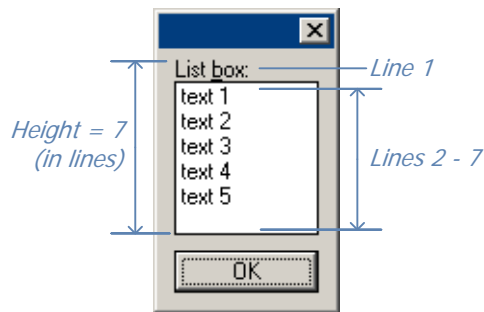
When this attribute is set to false, then the **value** attribute is restricted to the first digit. For example, **value = "1 2"** becomes **"1"**.



*Left: Selecting random items with the **Ctrl** key held down.
Right: Selecting sequential items with the **Shift** key held down.*

Height

The **height** attribute determines the height of the list box in lines. For example, **height = 7** means that the list box is seven lines tall, but has room for only six items, because the seventh line is used for the label.



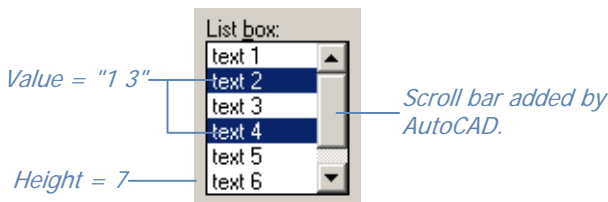
When **height** is set to **0** or is not included, then the list box is stretched to accommodate all items in the list, if possible.

Width

The **width** attribute determines the width of the list box. Width is measured in characters.

Other Attributes

The remaining attributes operate identically to those in other tiles. The list box illustrated below was created using the following DCL code:



```
: list_box {
    label = "List box.";
    mnemonic = "b";
    list = "text 1\ntext 2\ntext 3\ntext 4\ntext 5\ntext 6\ntext 7";
    value = "1 3";
    multiple_select = true;
    height = 7;
}
```

Popup_List

The **popup_list** tile displays a droplist. Despite the name, this tile drops down; it doesn't pop up.



```
: popup_list {
  action = "(LISP function)";
  key = "text";

  label = "text";
  mnemonic = "char";

  list = "text 1\ntext 2\ntext 3";
  tabs = "number number number";
  tab_truncate = false | true;
  value = "text";
  fixed_width_font = false | true;
  edit_width = 1-256;

  is_enabled= true | false;
  is_tab_stop= true | false;

  height = number;
  width = number;
  fixed_height = false | true;
  fixed_width = false | true;

  alignment = left | right | centered;
}
```

Label

The **label** attribute provides the prompt text for the droplist.

```
label = "Popup list: ";
```

Mnemonic

As with other tiles, you can specify the **Alt**+shortcut with the **&** prefix, or else use the **mnemonic** attribute to indicate the shortcut keystroke.

```
label = "&Popup list: ";
mnemonic = "P";
```

List

The **list** attribute specifies the text in the droplist tile. Each item is separated by the **\n** meta-character. When the list becomes too long for the droplist, progeCAD automatically adds a scroll bar.

```
list = "text 1\ntext 2\ntext 3";
```

Tabs

If you need text to line up in columns, use the **tabs** attribute to specify the tab spacing.

```
tabs = "10 20 30";
```

Then use the **\t** metacharacter to specify where the tabs occur in the **list** attribute.

```
list = "text 1\ttext 2\ttext 3";
```

Tab_Truncate

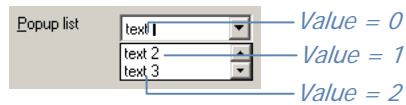
The **tab_truncate** attribute determines whether text is truncated when longer than the associated tab stop. Default is false, which means it is not truncated.

```
tab_truncate = true;
```

Value

The **value** attribute specifies which item is initially selected. The first item is #0 (the default). Use **value = ""** for no initial selection.

```
value = "1";
```



Other Attributes

The other attributes are identical to those described for other tiles.

Slider

The **slider** tile defines vertical and horizontal sliders.



```
: slider {  
  action = "(LISP function)";  
  key = "text";  
  
  label = "text";  
  mnemonic = "char";  
  
  layout = horizontal | vertical;  
  max_value = integer;  
  min_value = integer;  
  big_increment = integer;  
  small_increment = integer;  
  value = "text";  
  
  height = number;  
  width = number;  
  fixed_height = false | true;  
  fixed_width = false | true;  
  
  alignment = left | right | centered;  
}
```

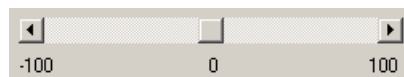
Label & Mnemonic

The **label** and **mnemonic** attributes seem to have no effect; they do not appear with the slider. The workaround is to use the **boxed_row** attribute to give the slider its label, as illustrated below:



```
: boxed_row {  
  label = "Slider: "; mnemonic = "S";  
  : slider {  
    max_value = 100;  
    min_value = -100;  
    big_increment = 10;  
    small_increment = 1;  
    value = "0";  
  }  
}
```

In addition to not labeling the slider, this tile provides no way to indicate to users the meaning of the minimum and maximum values. The workaround is to add a row of text underneath the slider, as illustrated here:



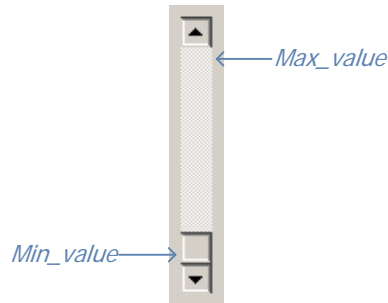
Notice that you need to use the **spacer** tile to position the three pieces of text appropriately:

```
: row {  
  : text { value = "-100"; alignment = left; }  
  : spacer {width = 11; }  
  : text { value = "0"; alignment = centered; }  
  : spacer {width = 8; }  
  : text { value = "100"; alignment = right; }  
}
```

Layout

The **layout** attribute determines if the slider is horizontal (default) or vertical, as illustrated below.

```
layout = vertical;
```



Horizontal sliders don't need to have a height or width attribute, because the default values are just fine. Vertical sliders, need the height specified, otherwise they end up with no height, as illustrated below. I suggest setting **height = 10**.



Using both a horizontal and vertical slider lets you create scroll bars for panning images.

Max_Value

The **max_value** attribute specifies the upper limit of the scroll bar; default = 10000. It limits the maximum value when the slider is at the right (or top) end of the bar. You can use any integer between -32768 and 32767. If you need larger values, then use LISP code to multiply them.

```
max_value = 32767;
```



Min_Value

The **min_value** attribute specifies the lower limit of the scroll bar; default = 0. It limits the minimum value when the slider is at the left (or bottom) end. You can use any integer between -32768 and 32767. To reverse the action of the scroll bar, use a larger value for **min_value** and a smaller one for **max_value**.

```
min_value = -32768;
```

Big_Increment

The **big_increment** attribute specifies the value of clicking the bar on either side of the slider. The default is 0.1 of the range between **max_value** and **min_value**. You can use any integer between the values of those two attributes.

Click here for Big_increment



```
big_increment = 100;
```

Small_Increment

The **small_increment** attribute specifies the value of clicking the arrows. The default is 0.01 of the range between **max_value** and **min_value**.

Click here...



...or here for Small_increment

```
small_increment = 1;
```

Value

The **value** attribute specifies the slider's initial position. Even though the value is an integer, it must be enclosed in quotation marks. The default is the same as **min_value**.

```
value = "1000";
```

Height

The **height** attribute specifies the size of vertical sliders; it has no effect on horizontal sliders. Height is measured in lines (of text). You have to specify a height for vertical sliders to avoid the problem described on the previous page.

```
height = 10;
```

Width

The **width** attribute specifies the size of horizontal sliders; it has no effect on vertical sliders. Width is measured in character (of text). You don't have to specify a width for horizontal sliders, because the default is satisfactory.

```
width = 40;
```

Fixed_Height & Fixed Width

The **fixed_height** and **fixed_width** attributes prevent DCL from expanding the slider to fit available space in the dialog box. Default in both cases is false, which means the height and width are not fixed. I suspect these attributes actually have no effect.

```
fixed_height = true;
```

```
fixed_width = true;
```

Alignment

The **alignment** attribute is supposed to shift the slider bar left or right, but I don't see that this attribute has any effect. The default is centered.

```
alignment = right;
```

Text

The **text** tile displays text in the dialog box. The text is static when specified in the DCL file, or dynamic when specified in the LSP file.

Label — Text

```
: text {  
  label = "text";  
  is_bold = false | true;  
  
  value = "text";  
  key = "text";  
  
  height = number;  
  width = number;  
  fixed_height = false | true;  
  fixed_width = false | true;  
  
  alignment = left | right | centered;  
}
```

Label

The **label** attribute specifies the text displayed by the dialog box. Autodesk recommends using this attribute for *static* text — text that doesn't change.

```
label = "Text";
```

Value

The **value** attribute also specifies text displayed by dialog box. Autodesk recommends you use this attribute for *dynamic* text — text that's specified by the accompanying LISP code. For dynamic text, **value** is set to *null*, as shown here:

```
value = "";
```

Make sure you include the **width** attribute so that there is sufficient space for the text message. progeCAD does not wrap text that is too long for the dialog box; text is truncated. And include the key attribute so that the LISP code can identify the text tile.

```
: text {  
  value = "";  
  key = "textField1";  
  width = 40;  
}
```

To display error messages or feedback on users' choices, use the **set_tile** function to assign text to the tile in the LISP code, like this:

```
(set_tile "textField1" "Error: Cannot set that value.")
```

The combination of DCL and LSP code results in the following display by the dialog box:

Error: Cannot set that value.

TIP If both **label** and **value** are used in the **text** tile code, however, then **value**'s text is displayed by the dialog box.

is_bold

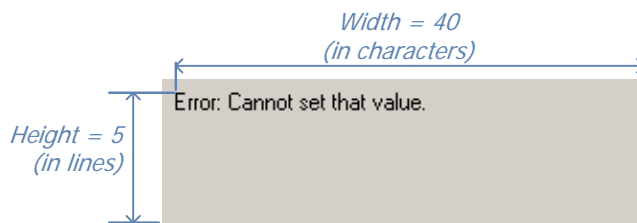
The **is_bold** attribute is supposed to make the text boldface, but I can see no difference.

```
is_bold = true;
```

height & width

The height and width attributes size the text tile. Height starts measuring from the top of the text, and is measured in lines. Width starts from the left end of the text, and is measured in characters.

```
height = 5;  
width = 40;
```



Fixed_Height & Fixed Width

The **fixed_height** and **fixed_width** attributes prevent DCL from expanding the text area to fit available space in the dialog box. Default in both cases is false, which means the height and width are not fixed.

```
fixed_height = true;  
fixed_width = true;
```

Alignment

The **alignment** attribute shifts the text to the left, right, or center of its width.

```
alignment = right;
```

Text_Part

The **text_part** tile displays text without margins, the blank space around text. (Well, it's supposed to, but as the example shows below, rather large gaps can occur.) It is meant to combine several pieces of text into one line, when used with the **concatenation** tile.

```
: text_part {  
    label = "text";  
}
```

Concatenation

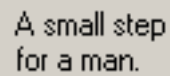
The **concatenation** tile strings together two or more **text** and/or **text_part** tiles.

A rectangular box containing the text "A small step" followed by "for a man." with a space between them.

```
: concatenation {  
  : text_part { label = "A small step"; }  
  : text_part { label = "for a man."; }  
}
```

Paragraph

The **paragraph** tile stacks lines of text, as illustrated below.

A rectangular box containing the text "A small step" stacked above "for a man." with a space between them.

```
: paragraph {  
  : text_part { label = "A small step"; }  
  : text_part { label = "for a man."; }  
}
```

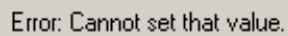
Errtile

The **errtile** tile defines a horizontal space for reporting error messages. It appears at the bottom of dialog boxes, and its key is “error.”

errtile;

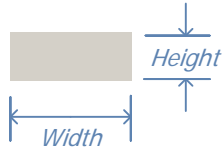
You use it in conjunction with the **set_tile** function in the accompanying LISP code.

```
(set_tile "error" "Error: Cannot set that value.")
```

A rectangular box containing the text "Error: Cannot set that value." with a space between "Error:" and "Cannot set that value.".

Spacer

The **spacer** tile defines a vertical and/or horizontal space.



```
: spacer {  
  height = number;  
  width = number;  
  fixed_height = false | true;  
  fixed_width = false | true;  
  alignment = left | right | centered;  
}
```

Spacer_0

The **spacer_0** tile defines a variable-width space that spaces itself automatically.

```
spacer_0;
```

Spacer_1

The **spacer_1** tile defines a very narrow space.

```
spacer_1;
```

Image

The **image** tile defines a rectangular area for displaying an image, such as of a text font sample, hatch pattern sample, color sample, or icons representing drawing and editing commands. The easiest is the color sample, as illustrated below, because it is specified by the **color** attribute.



Images of hatch patterns, fonts, and so on are placed on the image tile through the accompanying LISP code's callback function (**set_tile**) and the **key** attribute. There are two sources of image you can use:

- SLD slide files, which are created ahead of time with progeCAD's **MSlide** command, and then placed with LISP's **slide_image** function.
- Vector lines, which are drawn on-the-fly by LISP's **vector_image** function.

```
: image {
  action = "(LISP function)";
  key = "text";

  value = "text";
  mnemonic = "char";
  color = colornumber;

  aspect_ratio = number;
  height = number;
  width = number;

  is_enabled= true | false;
  is_tab_stop= true | false;
  fixed_height = false | true;
  fixed_width = false | true;

  alignment = left | right | centered;
}
```

Key

The **key** attribute identifies the **image** tile to the accompanying LISP code, so that the slide image can be placed in the dialog box.

```
key = "image1";
```

Value and Mnemonic

The **value** and **mnemonic** attributes appear to have no effect.

Color

The **color** attribute defines the color of the image tile. Use the same color numbers as for the **image_button** tile. A popular number is **-15**, which displays the same color as that of the dialog box's background — usually gray.

```
color = -15;
```

Number	Color Name	Meaning
0	black	ACI color 0 (black or white) ¹
1	red	ACI ² color 1
2	yellow	ACI color 2
3	green	ACI color 3
4	cyan	ACI color 4
5	blue	ACI color 5
6	magenta	ACI color 6
7	white	ACI color 7 (white or black) ¹

- 1graphics_foreground Current default color of entities (usually ACI 7). ¹
- 2graphics_background Current background color of progeCAD's graphics screen.
- 3blue
- 4black
- 5gray
- 6black
- 7red
- 15 dialog_background Current color of dialog box background (usually gray).
- 16 dialog_foreground Current color of dialog box text (usually black).
- 18 dialog_line Current color of dialog box lines (usually black).

Notes:

- ¹ The color is white when the background color is dark, but black when the background is light.
- ² ACI is short for "AutoCAD Color Index," and refers to the 256 color numbers.

Aspect_Ratio, Height, & Width

You use any two of these three attributes. The **aspect_ratio** attribute specifies the ratio between the height and width of the image tile, and must be used with either the **height** or the **width** attribute — but not both. Similarly, if you use the **height** and **width** attributes, you cannot use the **aspect_ratio** attribute. Examples:

```
aspect_ratio = 1.333;
height = 3;
```

Or...

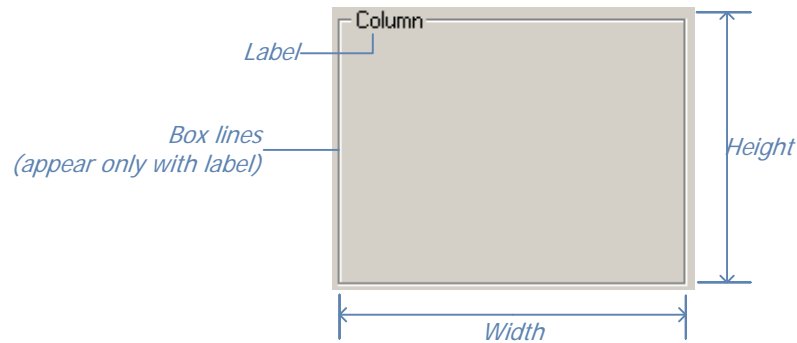
```
aspect_ratio = 1.333;
width = 4;
```

Or...

```
height = 3;
width = 4;
```


Column

The **column** tile defines a vertical column of tiles. This tile is not normally needed, because tiles are stacked vertically by default. You would use it when you want two columns of tiles in the dialog box.



```
: column {  
    label = "text";  
    height = number;  
    width = number;  
    fixed_height = false | true;  
    fixed_width = false | true;  
    children_fixed_height = false | true;  
    children_fixed_width = false | true;  
    alignment = left | right | centered;  
    children_alignment = left | right | centered;  
}
```

Label

The **label** attribute provides a title for the column. Curiously, when the label is not used, then the column is unboxed; when a label is used, the column is boxed — as if it were the **boxed_column** tile.

```
label = "Column";
```

Height & Width

progeCAD normally sizes the column automatically. You can use the **height** and **width** attributes to specify a larger size; height is measured in lines of text, width in characters.

```
height = 10;  
width = 40;
```

Children_Fixed_Height, Children_Fixed_Width, & Children_Alignment

The **children_fixed_height** and **children_fixed_width** attributes fix the height and width of clustered tiles; these attributes can be overridden by the children's attributes.

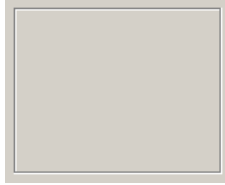
```
children_fixed_height = true;  
children_fixed_width = true;
```

The **children_alignment** attribute sets the alignment of clustered tiles; this attribute can be overridden by the children's alignment attributes.

```
children_alignment = centered;
```

Boxed_Column

The **boxed_column** tile places a box around a column of tiles. It is identical to the **column** tile, except that the box appears whether or not the tile has a label. The figure below illustrates the box without a label.



```
: boxed_column {  
  label = "text";  
  
  height = number;  
  width = number;  
  children_fixed_width = false | true;  
  fixed_height = false | true;  
  fixed_width = false | true;  
  children_fixed_height = false | true;  
  
  alignment = left | right | centered;  
  children_alignment = left | right | centered;  
}
```

Radio-Column & Boxed_Radio_Column

The **radio_column** and **boxed_radio_column** tiles define vertical columns for radio buttons. The only difference from the **boxed_column** and **column** tiles is the addition of the **value** attribute, which specifies which radio button is turned on.

```
: radio_column {           // or : boxed_radio_column {  
  label = "text";  
  value = "number";  
  
  height = number;  
  width = number;  
  fixed_height = false | true;  
  fixed_width = false | true;  
  children_fixed_height = false | true;  
  children_fixed_width = false | true;  
  
  alignment = left | right | centered;  
  children_alignment = left | right | centered;  
}
```

Value

The **value** attribute specifies which radio button is turned on, the first button being #0.

```
value = "2";
```

Row & Boxed_Row

The **row** and **boxed_row** tiles define a horizontal row of other tiles, called “children” or “clustered tiles.” Like columns, including a label to the row tile adds the box; no label, no box. Otherwise, the two tiles are identical.



```
: row {                                     // or : boxed_row {
    label = "text";
    height = number;
    width = number;
    fixed_height = false | true;
    fixed_width = false | true;
    children_fixed_height = false | true;
    children_fixed_width = false | true;
    alignment = left | right | centered;
    children_alignment = centered | top | bottom;
}
```

Other Attributes

Attributes are identical to those of the column tile, except that the **children_alignment** attribute is vertically oriented: top, bottom, or centered.

Radio_Row & Boxed_Radio_Row

The **radio_row** tile defines a horizontal row of radio buttons.

```
: radio_row {                               // or : boxed_radio_row {
    label = "text";
    value = "number";

    height = number;
    width = number;
    fixed_height = false | true;
    fixed_width = false | true;
    children_fixed_height = false | true;
    children_fixed_width = false | true;
    alignment = left | right | centered;
    children_alignment = centered | top | bottom;
}
```

Value

The **value** attribute specifies which radio button is turned on, the first button being #0.

```
value = "2";
```

OurBase.Dcl

The *ourbase.dcl* file defines common prototypes and subassemblies for use by progeCAD and user-defined dialogs. This file is found in the C:\Program Files\progeSOFT\progeCAD 2009 Pro ENG\api\dcl folder.

Copyright (c) 1998 by Visio Corporation. All rights reserved.
The Software is subject to the license agreement that accompanies or is included with the Software, which specifies the permitted and prohibited uses of the Software. Any unauthorized duplication or use of Visio Corporation Software, in whole or in part, in print, or in any other storage and retrieval system is prohibited.
To the maximum extent permitted by applicable law, Visio Corporation and its suppliers disclaim any and all warranties and conditions, either express or implied, including, without limitation, implied warranties of merchantability, fitness for a particular purpose, title, and non-infringement, and those arising out of usage of trade or course of dealing, concerning these materials. These materials are provided "as is" without warranty of any kind.

```
base_about : dialog {
  label = "About BASE";
  /*
  : paragraph {
    alignment = left;
    fixed_height = true;
    : text {
      label = "Copyright (c) 1997,Visio Corporation";
    }
    : text {
      label = "All Rights Reserved.";
    }
    : concatenation {
      : text_part {
        label = "Version ";
      }
      : text_part {
        label = "";
        key = "version";
        width = 50;
      }
    }
  }
  : radio_row {
  key="row1";
  : radio_button {
    label="ALP01";
    key="alpo1";
  }
  : radio_button {
    key="alpo2";
    label="ALP02";
  }
  : radio_button {
    key="alpo3";
    label="ALP03";
  }
  : radio_button {
    label="ALP04 ";
    key="alpo4";
  }
  }
  spacer_1;
  /*
  ok_cancel;
}
```

LISP Functions for Dialog Boxes

Dialog boxes are designed by DCL files and displayed by LISP routines. The most basic LISP routine to load, display and unload dialog boxes looks like this:

```
(defun c:funcName (/ dlg-id)
  (setq dlg-id (load_dialog "fileName"))
  (new_dialog "dialogName" dlg-id)
  ; Insert get_tile, set_tile, action_tile, and other functions here.
  (start_dialog)
  (unload_dialog dlg-id)
)
```

The *fileName.dcl* file specifies the layout of the dialog box. The most basic file looks like this:

```
dialogName : dialog {
  // Insert tiles here.
  ok_only;
}
```

This section of the chapter describes the LISP functions that interact with dialog boxes, in the following order:

```
load_dialog
  new_dialog
  start_dialog
  done_dialog
  term_dialog

get_tile
  set_tile
  get_attr
  mode_tile
  action_tile
  client_data_tile

start_list
  add_list
  end_list

start_image
  slide_image
  fill_image
  vector_image
  dimx_tile
  dimy_tile
  end_image

alert
  help
  acad_helpdlg
  acad_colordlg
  acad_truecolordlg
  initdia
```

Alphabetical List of LISP Functions for Dialog Boxes

action_tile	Assigns action to be evaluated when user selects dialog box tile.
add_list	Adds and modifies strings in current dialog box listbox.
client_data_tile	Associates data from an application with a tile in the dialog box.
dimX_tile	Returns the x-dimension of the dialog box image tile.
dimY_tile	Returns the y-dimension of the dialog box image tile.
done_dialog	Terminates the dialog box.
end_image	Ends creation of dialog box image tile.
end_list	Ends processing of dialog box list box.
fill_image	Draws filled rectangles in dialog boxes.
get_attr	Retrieves the DCL value of the tile's attribute.
get_tile	Retrieves the value of tile.
load_dialog	Loads .dcl files that define dialog boxes.
mode_tile	Sets the mode of dialog box tiles.
new_dialog	Activates dialog boxes.
set_tile	Sets the value of dialog box tiles.
slide_image	Displays slides in dialog box image tiles.
start_dialog	Displays the current dialog box.
start_image	Starts creating images in dialog boxes.
start_list	Starts processing lists in dialog boxes.
term_dialog	Terminates dialog boxes.
unload_dialog	Unloads .dcl files from memory.
vector_image	Draws vectors in dialog box image tiles.

Dialog Boxes Displayed by LISP Functions

acad_colorldg	Displays the Select Color dialog box with only the Index Color tab.
acad_truecolorldg	Displays the Select Color dialog box with all tabs.
alert	Displays the alert dialog box with customized warning.
help	Displays the Help window.
initdia	Forces display of the next command's dialog box.

Load_Dialog

The **load_dialog** function loads *.dcl* files that define dialog boxes, and returns a *fileid* (the identifying number assigned by the operating system to open files) handle.

```
(load_dialog dclFile)
```

dclFile — name of the *.dcl* file. It is in quotation marks; remember to use double-slash path separators, as shown below. The “.dcl” extension is not required.

```
(load_dialog "c:\\filename")
```

This function is usually used with **setq** to store the value of the handle, as follows:

```
(setq dclId (load_dialog "c:\\filename"))
```

This function returns a fileid handle such as 30, when successful, or **-1** if not.

New_Dialog

The **new_dialog** function activates a named dialog box. This function is needed because *.dcl* files can contain more than one dialog box definition. Thus, **load_dialog** is used to load the *.dcl* file, and then **new_dialog** is used to access the specific dialog box.

```
(new_dialog dlgName dclId action screenPt)
```

dlgName — string identifying the dialog box in the *.dcl* file.

dclId — DCL fileid handle retrieved earlier by the **load_dialog** function.

action — [optional] string containing the LISP expression that executes as default action when users pick tiles that don't have a DCL action or LSP callback assigned by the **action_tile** function.

screenPt — [optional] 2D point list specifying the x,y-location of the dialog box's upper left corner of the progeCAD window. Use '**(-1 -1)**' to open the dialog box in the center of the progeCAD window. To use this argument without the action argument, enter "", as follows:

```
(new_dialog dlgName dcl_id "" (10,10))
```

This function returns **T** when successful, or **nil** if not.

Start_Dialog

The **start_dialog** function displays the dialog box. Before this function is executed, you should set up callbacks and other functions. This function has no arguments.

```
(start_dialog)
```

This function returns **1** when users exit the dialog box by clicking OK, or **0** if they click the Cancel button. A **-1** is returned when the dialog box is closed by the **term_dialog** function.

Done_Dialog

The **done_dialog** function closes the dialog box.

(**done_dialog** *status*)

status — positive integer returned by **start_dialog**, the meaning of which the application determines. For this to work, **done_dialog** must be called from a callback function such as **action_tile**.

This function returns a 2D point list in the form of '(x,y). It identifies the position of the upper-left corner of the dialog box at the time the user exited it. This allows you to reopen the dialog box in the same location.

Term_Dialog

The **term_dialog** function terminates dialog boxes. It is called by progeCAD when applications (LISP routines) terminate while *.dcl* files are still open. This function has no arguments.

(**term_dialog**)

This function always returns **nil**.

Unload_Dialog

The **unload_dialog** function unloads *.dcl* files from memory.

(**unload_dialog** *dclId*)

dclId — specifies the fileid handle first acquired by the **load_id** function.

This function always returns **nil**.

Get_Tile

The **get_tile** function retrieves the values of tiles.

(get_tile key)

key — identifies the tile to be accessed.

This function returns a string containing the value of the tile's **value** attribute.

Set_Tile

The **set_tile** function sets the value of dialog box tiles.

(set_tile key value)

key — identifies the tile to be processed.

value — specifies a string that contains the new value to be assigned to the tile's **value** attribute.

This function returns the new value of the tile.

Get_Attr

The **get_attr** function retrieves the DCL value of the tile's attribute.

(get_attr key attribute)

key — identifies the tile to be processed.

attribute — identifies the attribute whose value is to be retrieved.

This function returns a string with the attribute's as found in the DCL file.

Mode_Tile

The **mode_tile** function sets the mode of dialog box tiles. This allows you to change, for example, buttons from active (normal) to inactive (grayed out).

(mode_tile key mode)

key — identifies the tile to be processed.

mode — specifies the action to be applied to the tile:

Mode	Meaning
0	Enables the tile.
1	Disables the tile (grays it out).
2	Sets focus to the tile.
3	Selects the contents of the edit box.
4	Toggles image highlighting.

This function returns **nil**.

Action_Tile

The **action_tile** function assigns action to be evaluated when users select the dialog box's tile.

(**action_tile** *key action*)

TIP The action assigned by this function overrides the action defined by the tile's **action** attribute, as well as the action specified by the **new_dialog** function.

key — identifies the tile to be processed.

action — a string that specifies the action, usually an LISP function. (LISP's **command** function cannot be used, unfortunately.) You can use the following metacharacters:

Metacharacter	Meaning
\$value	Current value of the tile.
\$key	Name of the tile.
\$data	Application-specific data set by client_data_tile .
\$reason	Callback reason.
\$x and \$y	Image's x,y coordinates.

This function returns **T**.

Client_Data_Tile

The **client_data_tile** function associates data from a function with a tile in the dialog box.

(**client_data_tile** *key data*)

key — identifies the tile to be processed.

data — specifies the string containing the data.

TIP Functions can refer to this data as **\$data**.

This function returns **nil**.

Start_List

The **start_list** function starts processing list boxes and popup boxes.

(**start_list** *key operation index*)

key — identifies the list box or popup box being processed.

operation — [*optional*] specifies the operation being performed; default is to delete the existing list, and replace it with a new one specified by the **add_list** function. The operations are:

Operation	Meaning
1	Change selected list contents
2	Append new list entry
3	Delete old list and create new list (the default)

index — [*optional*] specifies which list item to modify; default is #0, the first item.

This function returns the name of the list.

TIPS In all cases, you use the list-related functions in this order:
(**start_list**)
(**add_list**)
(**end_list**)

Autodesk warns against using the **set_tile** function between **start_list** and **end_list**, because that would change the nature of the list.

All actions by the **add_list** function apply only to the list specified by **start_list**; to switch to a different list, use **end_list** and then **start_list**.

Add_List

The **add_list** function adds or modifies strings in list and popup boxes, depending on the operation specified by **start_list**.

(**add_list** *strings*)

strings — specifies the list of items to add or replace in the list. The string uses quotation marks to separate items in the list, as follows:

(**add_list** "firstItem" "secondItem" "thirdItem")

This function returns the string, if successful; otherwise **nil**, if not.

End_List

The **end_list** function ends processing of list and popup boxes.

(**end_list**)

This function returns **nil**.

Start_Image

The **start_image** function starts creating vector or slide images in dialog boxes.

`(start_image key)`

key — specifies the key name of the image tile.

This function returns the value of *key*; otherwise **nil**, if unsuccessful.

TIPS Typically, you use the image-related functions in this order:
`(start_image)`
`(fill_image)`
`(slide_image)` or `(vector_image)`
`(end_image)`

Slide_Image

The **slide_image** function displays slides in dialog box image tiles.

`(slide_image x y width height sldName)`

x — specifies the number of pixels to offset the image from the upper-left corner of the tile, in the x direction.

y — specifies the number of pixels to offset the image from the upper-left corner of the tile, in the y direction.

width — specifies the width of the image in pixels.

height — specifies the height of the image in pixels.

sldName — specifies the name of the slide image to display, which can be in an SLD slide file or SLB slide library file. When in a library, use this format:

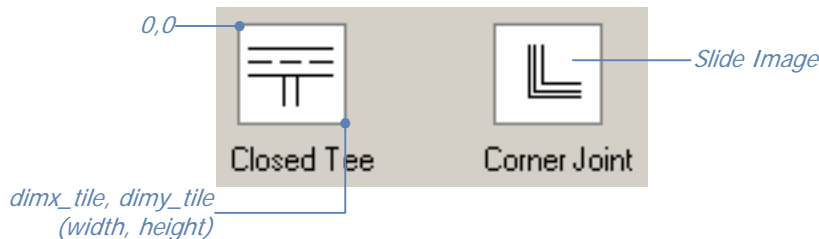
`(slide_image 0 0 40 30 sldlibName(sldName))`

TIPS X and Y are always positive.

The coordinates of the upper left corner are 0,0.

You can get the coordinates of the lower-right corner through **dimx_tile** and **dimy_tile**, like this:

`(slide_image 0 0 (dimx_tile "slide_tile") (dimy_tile "slide_tile") "sldName")`



This function returns the name of the *sldName* as a string.

Fill_Image

The **fill_image** function draws filled rectangles in dialog boxes.

(fill_image x y width height color)

x — specifies the number of pixels to offset the image from the upper-left corner of the tile, in the x direction.

y — specifies the number of pixels to offset the image from the upper-left corner of the tile, in the y direction.

width — specifies the width of the image in pixels.

height — specifies the height of the image in pixels.

color — specifies the color using ACI, or one of the following special numbers:

Number	Meaning
-2	Current background color of progeCAD's drawing area.
-15	Current background color of the dialog box.
-16	Current text color of the dialog box.
-18	Current color of dialog box lines.

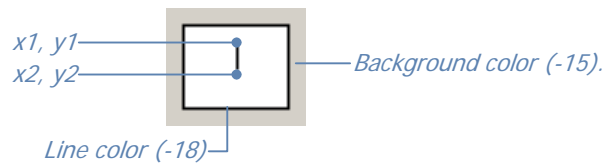
This function returns an integer representing the ACI fill color.

TIP This function must be used between the **start_image** and **end_image** functions.

Vector_Image

The **vector_image** function draws vectors in dialog box image tiles.

(vector_image x1 y1 x2 y2 color)



x1 — specifies the x coordinate of the starting point.

y1 — specifies the y coordinate of the starting point.

x2 — specifies the x coordinate of the starting point.

y2 — specifies the y coordinate of the starting point.

color — specifies the color using ACI, or one of the special numbers listed above.

TIP One vector (line) is drawn with each call of this function. The line is drawn from $x1, y1$ to $x2, y2$.

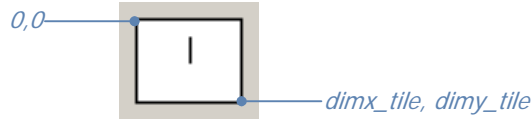
DimX_Tile & DimY_Tile

The **dimx_tile** function returns the x-dimension of the image tile's lower right corner; the **dimy_tile** function does the same for the y-dimension.

(dimx_tile key)

(dimy_tile key)

key — specifies the key name of the image tile.



These functions return the “x-1” width and “y-1” height of the tile.

TIPS *Caution:* These functions return x,y coordinates are one less than the total x and y dimensions of the tile, because the upper-right corner is 0,0.

These functions are meant for use with the **slide_image**, **fill_image**, and **vector_image** functions.

End_Image

The **end_image** function signals the end of the image tile's definition.

(end_image)

This function returns **nil**.

Dialog Boxes Displayed by LISP Functions

The following LISP functions display progeCAD dialog boxes.

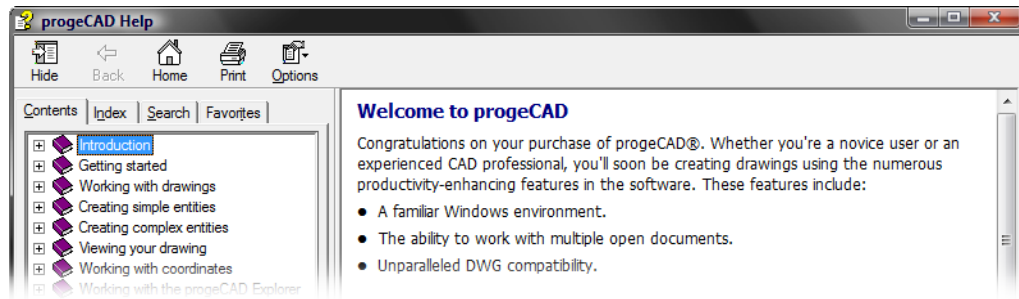
Alert

The **alert** function displays the alert dialog box with customized warning. You can use the `\n` metacharacter to include line breaks.

```
(alert "Help me!\nI've fallen and I can't get up!")
```

Help

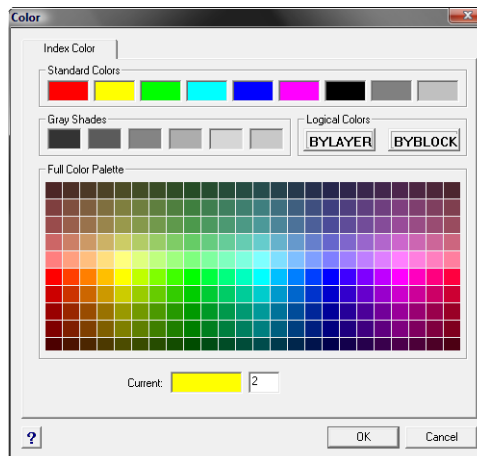
The **help** function displays the Help window.



AcadColorDlg

The **acad_colorDlg** function displays the Select Color dialog box with just the Index Color tab.

```
(acad_colorDlg colorNum flag)
```



colorNum — specifies the default color number; ranges from 0 to 256. This integer is a required argument, even when you don't want to specify a default.

0 = ByBlock color.

256 = ByLayer color.

flag — [optional] disables the **ByLayer** and **ByBlock** buttons when set to **nil**.

For example, to open the Select Color dialog box, set red (1) as the default color, and gray out the By-buttons, use this form of the function:

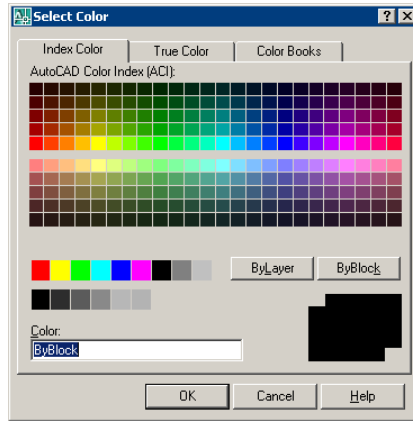
```
(acad_colordlg 1 nil)
```

This function returns the number of the color selected by the user, or nil when the user clicks Cancel.

Acad_TrueColorDlg

The **acad_truecolordlg** function displays the Select Color dialog box with all tabs.

```
(acad_truecolordlg color flag byColor)
```



color — specifies the the default color as a dotted pair, where the first value is the DXF code for the type of color specification:

62 = ACI (index color).

420 = TrueColor spec in RGB (red-green-blue) format.

430 = color book name.

Use the following formats:

Color Format	Dotted Pair Format	Example for Red
ACI	(62 . ColorIndex)	(62 . 1)
TrueColor	(420 . "red,green,blue")	(420 . "255,0,0")
Color Book	(430 . "colorbook\$colorname")	(430 . "RAL CLASSIC\$RAL 3026")

flag — [optional] disables the **ByLayer** and **ByBlock** buttons when set to **nil**.

byColor — [optional] sets the value of ByLayer and ByBlock color; use the same format as for *color*.

This function returns the color selected by the user in dotted-pair format. The list may contain more than one dotted-pair; the last one is the one selected by the user. For example, if the user selects from a color book, then the list contains the 430 pair (specifying the color book), as well as a 420 pair containing the TrueColor value and a 62 pair describing the closest ACI value.

Nil is returned when the user clicks **Cancel**.

InitDia

The **initdia** function forces the display of the dialog box of the following command, such as: x.

```
(initdia flag)
(command "image")
```

flag — [*optional*] when 0, resets command to display prompts at the command line.

This function is meant for commands that normally display their prompts at the command line during LISP routines. This function always returns *nil*.

Employing Diesel Expressions

With a name like “direct interactively evaluated string expression language,” the programming logic of Diesel is as clear as the acronym’s meaning. Despite the word “string” (meaning text), Diesel also operates with numbers.

The original purpose of the Diesel macro language was to customize the AutoCAD status bar, but it has since found its way into menu and toolbar macros, and became the most powerful programming environment available in AutoCAD LT — much to the chagrin of Autodesk, who had deliberately disabled the AutoLISP that was to ship with LT.

Is Diesel a Programming Language?

For me, the line of demarkation macros and true programming languages is whether they contains logic functions, such as *if-then*, *while*, *greaterthan*, and so on. (Logic functions allow programs to make decisions.)

Diesel has logic functions. But overall, its syntax is so obscure and functions so few that it begs to be known as a macro language — and that’s how I’ll refer to it from now on.

TIP At time of writing this book, progeCAD did not support accessing system variables with the \$(getvar function.

In This Chapter

- How Diesel works.
- Diesel function names.
- Using the ModeMacro and MacroTrace commands.
- Using Diesel in macros and LISP.
- Debugging Diesel.

What Diesel Does

Can make changes to the progeCAD's status bar so that it displays information useful to you, such as the current elevation, the *.dwg* file name, or the current time. There is a limitation, however: the text displayed by Diesel is truncated after a 39 characters, no matter how big you make the progeCAD window. To display text on the status bar, use the **ModeMacro** command.

Diesel has an unusual format for a macro language. Every function begins with a dollar sign and a bracket:

```
$(function,variable)
```

No doubt, the purpose of the \$-sign is to alert progeCAD's command processor that a Diesel expression is on the way, just as the (symbol alerts progeCAD that an LISP expression is coming up. (The \$ symbol is sometimes used by programmers to indicate strings.)

The beginning and end of Diesel functions are indicated by opening and closing parentheses. Parentheses also allow Diesel functions to be *nested*, where the result of one function is evaluated by a second. Parentheses allow Diesel to work on more than one variable at a time — up to nine variables for some functions; the closing parenthesis alerts Diesel to the end of the list of variables.

In total, there are 28 functions available in Diesel. Most of them take at least one variable, some as many as nine. Functions can be used at the command line, in toolbar and menu macros, in LISP code, and other areas of progeCAD. A comma always separates the function name and its variable(s). *Diesel tolerates no spaces.*

Brief List of Diesel Functions

Here is a summary of the functions supported by Diesel:

Math Functions

+	Addition.
-	Subtraction.
•	Multiplication.
/	Division.

Logic Functions

=	Equal.
<	Less than.
>	Greater than.
!=	Not equal.
<=	Less than or equal.
>=	Greater than or equal.
and	Logical bitwise AND.
eq	Determines if all items are equal.
if	If-then.
or	Logical bitwise OR.
xor	Logical bitwise XOR.

Numeric Conversion Functions

angtos	Formats angles (angle to string).
fix	Truncates real numbers to rounded-down integers.
rtos	Formats numbers with units (real to string).

String (Text) Functions

index	Extracts one element from a comma-separated series.
nth	Extracts the <i>n</i> th element from one or more items.
strlen	Returns the number of characters in the string (string length).
substr	Returns a portion of a string.
upper	Converts a text string to uppercase characters.

System Functions

edtime	Formats the system time.
eval	Passes a string to Diesel.
getvar	Gets the value of a system variable.
getenv	Gets the value of an environment variable (does not operate in progeCAD).
linelen	Returns the length of the display (does not operate in progeCAD).

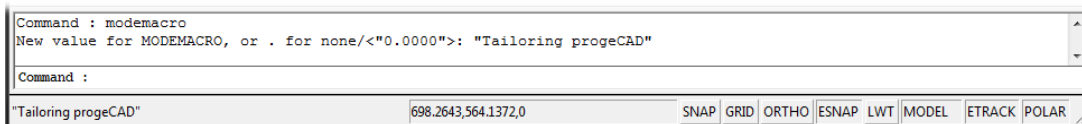
ModeMacro: Displaying Text on the Status Bar

The ModeMacro command is used to display text on the status bar. Let's see how that works.

1. Enter the **ModeMacro** system variable at the 'Command:' prompt, and then type something:

```
Command: modemacro  
New value for MODEMACRO, or . for none <"": Tailoring progeCAD
```

The words "Tailoring progeCAD" should appear next to the coordinate display:



(You cannot change the location of this text.)

2. To remove the text from the status bar, type the **ModeMacro** system variable with a . (null string), as follows:

```
Command: modemacro  
New value for MODEMACRO, or . for none <"Tailoring progeCAD">: ""
```

Reporting Values of System Variables

To display the values of system variables on the status bar, use Diesel's **\$(getvar)** function. This function gets the value of a system variable, and then displays it on the status bar.

1. For this tutorial, display the current elevation with the **FilletRad** system variable, as follows:

```
Command: modemacro  
New value for MODEMACRO, or . for none <"": $(getvar,filletrad)
```

progeCAD displays 0.0000 or something similar on the status bar.

2. Use the **FilletRad** system variable to change the elevation to 10:

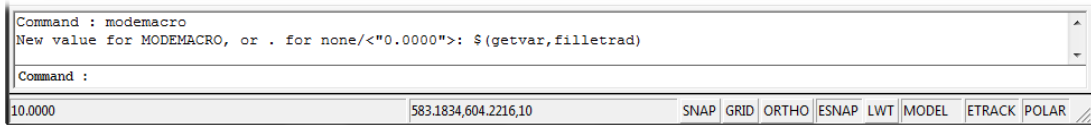
Command: **filletrad**

Enter new value for FILLETRAD <0.0000>: **10**

3. Reenter the ModeMacro sysvar to update the value reported on the status bar:

Command: **modemacro**

New value for MODEMACRO, or . for none <"0.0000">: **\$(getvar,filletrad)**

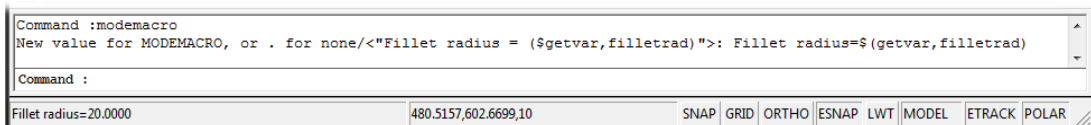


4. The values of 0 and 10 can be made more useful by adding a description, such as Fillet radius=10.” Here’s how:

Command: **modemacro**

New value for MODEMACRO, or . for none <"">: **Fillet radius=\$(getvar,elevation)**

Notice that text can be used together with Diesel code.



While Diesel can *get* the values of system variables, it cannot change them, because there is no related “setvar” function.

Debugging Diesel

Diesel is not a forgiving language, and so it is easy to make errors. Here are some common mistakes you might make:

- Forget the closing parenthesis:

`$(+,1,2` *should be* `$(+,1,2)`

- Forget the closing quotation mark on the right:

`$(upper,"to)` *should be* `$(upper,"to").`

- Enter an incorrect function name:

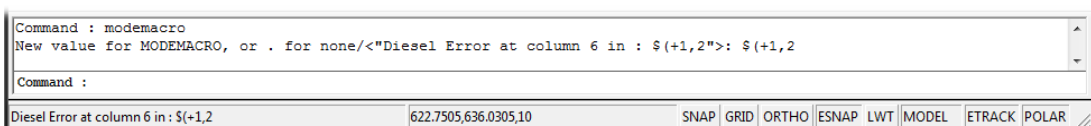
`$(stringlength,"Example")` *should be* `$(strlen,"Example").`

- Include the wrong number of arguments for the function:

`$(edtime)` *should be* `$(edtime,$(getvar,date),DDD).`

- Text to be displayed on the status bar has too many characters.

When progeCAD senses an error, it reports the location on the status bar, like this:



MacroTrace

To help track down bugs in Diesel macros, turn on the undocumented **MacroTrace** system variable, as follows:

```
Command: macrotrace  
New value for MACROTRACE <0>: 1
```

When on, progeCAD displays a step-by-step evaluation of the Diesel macro in the Text window. Here's how it works for the following Diesel macro, which converts the value of the fillet radius to metric: `$(*,2.54, $(getvar,filletrad))`

```
Command: modemacro  
New value for MODEMACRO, or . for none <"">: $(*,2.54,$(getvar,filletrad))  
Eval: $(*, 2.54, $(getvar,filletrad))  
Eval: $(GETVAR, filletrad)  
==> 0.5  
==> 1.27
```

There is a bug in MacroTrace that causes it to reevaluate the most recent Diesel expression over and over again. Each time I type something at the 'Command:' prompt (such as the Line command), MacroTrace re-displays its evaluation. It does not, however, interfere except visually. Turn off MacroTrace when you no longer need it, as follows:

```
Command: macrotrace  
New value for MACROTRACE <1>: 0
```

Using Variables

Diesel functions work with variables. First, use the **SetVar** command to manually store a value in one of the user system variables, such as **UserR1**:

```
Command: setvar  
Enter variable name or [?] <MODEMACRO>: userr1  
Enter new value for USERR1 <0.0000>: 3.141
```

And then access it with the `$(getvar)` function:

```
$(+,$(getvar,userr1),25)
```

You can use the following user system variables:

UserR1 through **UserR5** for storing real numbers (numbers with decimals).

UserI1 through **UserI5** for storing integers (numbers without decimals).

UserS1 through **UserS5** for storing strings (text).

Diesel Functions

Here are additional details on Diesel functions.

Math Functions

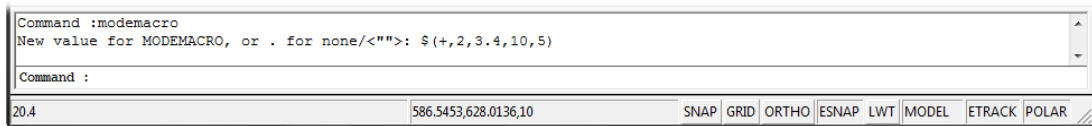
Diesel supports the four basic arithmetic functions.

+ (Addition)

The + function adds together up to nine numbers:

```
$(+,2,3.4,10,5)          returns          20.4
```

The answer appears on the status bar:



The function works with as little as one value:

```
$(+,2)                   returns          2
```

- (Subtraction)

The - function subtracts as many as eight numbers from a ninth. For example, the following equation should be read as $2 - 3.4 - 10 - 5 = -16.4$:

```
$(-,2,3.4,10,5)         returns          -16.4
```

And this equation should be read as $2 - 0 = 2$:

```
$(-,2)                   returns          2
```

* (Multiplication)

The * function multiplies together up to nine numbers.

```
$(*,2,3.4,10,5)         returns          340
```

When you have the value of pi (3.141) stored in **UserR1**, you can perform calculations that involve circles.

For instance, to find the area of a circle, recall that the formula is $\pi * r^2$. Diesel doesn't support squares or exponents, so you need to multiple **r** by itself: $\pi * r * r$. To find the area of a 2.5"-radius circle, enter the following:

```
$(*,$(getvar,userR1),2.5,2.5)  returns          19.63125
```

/ (Division)

The / function divides one number by up to eight other numbers.

```
$(/,2,3.4,10,5)         returns          0.01176471
```

This should be read as $2 \div 3.4 \div 10 \div 5 = 0.1176471$.

Logic Functions

The logic functions test to see if two (or more) values are equal (or not).

= (Equal)

The = function determines if two numbers (or strings) are equal. If so, the function returns 1; if not, it returns 0.

<code>\$(eq,2,2)</code>	<i>returns</i>	1
<code>\$(eq,2,3.4)</code>	<i>returns</i>	0

The values have to be exactly equal; for instance, a real number is not the same as an integer number, as the following example illustrates:

<code>\$(eq,2.0,2)</code>	<i>returns</i>	0
---------------------------	----------------	---

Normally, you would not test two numbers, but you would a number and a value stored in a variable. For example, to check if **LUnits** is set to 4 (architectural units):

<code>\$(eq,\$(getvar,lunits),4)</code>	<i>returns</i>	1	<i>when LUnits = 4.</i>
	<i>returns</i>	0	<i>if LUnits = any other number.</i>

< (Less than)

The < function determines if one number is less than another. If so, the function returns 1; if not, it returns 0.

<code>\$(<,2,2)</code>	<i>returns</i>	0
<code>\$(<,2,3.4)</code>	<i>returns</i>	1

> (Greater Than)

The > function determines if one number is greater than another. If so, the function returns 1; if not, it returns 0.

<code>\$(>,2,2)</code>	<i>returns</i>	0
<code>\$(>,2,3.4)</code>	<i>returns</i>	1

!= (Not Equal)

The != function determines if one number is not equal to another. If not equal, the function returns 1; if equal, it returns 0.

<code>\$(!=,2,2)</code>	<i>returns</i>	0
<code>\$(!=,2,3.4)</code>	<i>returns</i>	1

<= (Less Than or Equal)

The <= function determines if one number is less or equal than another. If so, the function returns 1; if not, it returns 0.

<code>\$(<=,2,2)</code>	<i>returns</i>	1
<code>\$(<=,2,3.4)</code>	<i>returns</i>	1
<code>\$(<=,9,0.5)</code>	<i>returns</i>	0

>= (Greater Than or Equal)

The `>=` function determines if one number is greater than or equal than another. If so, the function returns 1; if not, it returns 0.

<code>\$(>=,2,2)</code>	returns	1
<code>\$(>=,9,0.5)</code>	returns	1
<code>\$(>=,2,3.4)</code>	returns	0

and (Logical Bitwise AND)

The **and** function returns the bitwise logical “AND” of two or more integers. This function operates on up to nine integers.

eq

The **eq** function determines if two numbers (or strings) are equal. If identical, the function returns 1; otherwise, it returns 0.

<code>\$(eq,2,2)</code>	returns	1
<code>\$(eq,9,0.5)</code>	returns	0

This function appears to operate identically to the `=` function.

if

The **if** function checks if two expressions are the same. If so, the function carries out the first option, and ignores the second option; if false, it carries out the second option. In generic terms:

```
$(if,test,true,false)
```

where

test — specifies the logic function, such as `$(eq,clayer,0)`; the *test* expects a value of 1 (true) or 0 (false).

true — indicates the action to take when the test is true.

false — indicates the action to take when the test is false.

For example, the following test checks to see if the current layer is not 0. If so, it then gets the name of the layer. Notice that the *true* parameter is missing.

```
$(if,$(eq,clayer,"0"),$(getvar,clayer))
```

or (Logical Bitwise Or)

The **or** function returns the bitwise logical “OR” of two or more integers.

xor (Logical Bitwise Xor)

The **xor** function returns the bitwise logical “XOR” (eXclusive OR) of two or more integers.

Conversion Functions

The conversion functions change the state of numbers.

angtos

The **angtos** function formats numbers as angles (short for “angle to string”). In generic terms, the function looks like this:

```
$(angtos,value,mode,prec)
```

where:

value — the real number being formatted; an angle.

mode — determines the formatting; see table below.

prec — specifies the precision; see table below.

Mode and *prec* are optional; when left out, Diesel uses the values specified by the AUnits and AuPrec system variables, respectively.

Mode (AUnits)	Meaning
0	Displays angles as decimal degrees.
1	Displays angles as degrees, minutes, seconds.
2	Displays angles as grads.
3	Displays angles as radians.
4	Displays angles as surveyor's units.

Prec	Range (AuPrec)
Decimal	0 to 0.00000000
DMS	0d to 0d00'00.0000"
Grads	0g to 0.00000000g
Radians	0r to 0.00000000r
Surveyor's units	N 0d E to N 0d00'00.0000" E

You can preset the *mode* and *prec* with the **Units** command. For example, when the Units command is set to Surveyor's Units with a precision of 0 decimal places, we get the following response:

```
$(angtos,90) returns N 26d37'13" W
```

Or, we can use the optional *mode* and *prec* settings inside the AngToS function:

```
$(angtos,90,4,3) also returns N 26d37'13" W
```

fix

The **fix** function removes the decimal portion from real numbers, converting them to integers. This function can be used to extract the number before the decimal point from a real number. (There is no “round” function.)

```
$(fix,3.99) returns 3
```

rtos

The **rtos** function applies units to numbers (short for “real to string”). This can be useful for displaying two different measurement units on the status bar. This function operates similarly to the **angtos** function:

```
$(rtos,value,mode,prec)
```

where:

value — the real number being formatted.

mode — determines the formatting; see table below.

prec — specifies the precision; see table below.

Mode and *prec* are optional; when left out, Diesel uses the values specified by the LUnits and LuPrec system variables, respectively (L is short for “linear”).

Mode (LUnits)	Meaning
1	Displays numbers in scientific notation.
2	Displays numbers in decimal format.
3	Displays numbers in engineering format.
4	Displays numbers in architectural format.
5	Displays numbers in fractional format.

Prec	Range (LuPrec)
Scientific	0E+01 to 0.00000000E+01
Decimal	0 to 0.00000000
Engineering	0'-0" to 0'-0.00000000"
Architectural	0'-0" to 0'-0 1/256"
Fractional	0 to 0 1/256

You can preset the *mode* and *prec* with the **Units** command. When the Units command sets Architectural units with a precision of 3 decimal places, we get this result:

```
$(rtos,90.25) returns 7'-6 1/4"
```

Or, you can use the optional *mode* and *prec* settings inside the function:

```
$(rtos,90.25,4,3) also returns 7'-6 1/4"
```

String Functions

The string functions manipulate text (and sometimes numbers).

index

The **index** function extracts one element from a comma-separated series. Autodesk suggests using this function to extract the x, y, and z coordinates from variables returned by the (**\$getvar** function. In generic terms, the function looks like this:

```
$(index,item,string)
```

where:

item — a counter; starts with 0.

string — the text being searched; contains comma-separated items.

Note that the *item* counter starts with 0, instead of 1; the first item is #0:

```
$(index,0,"2,4,6") returns 2
```

String must be text surrounded by quotation marks; if you leave out the quotes, Diesel ignores the function. The string consists of one or more items separated by commas.

Here is an example of extracting the y coordinate from the **LastPoint** system variable:

```
$(index,1,$(getvar,lastpoint)) returns 64.8721
```

(The result will differ, depending on the coordinate stored in LastPoint.) Use the following *item* values to extract specific coordinates:

Item	Coordinate Extracted
0	X
1	Y
2	Z

nth

The **nth** function extracts the *n*th element from one or more items. This function handles up to eight items. Like **index**, the first item in the list is #0. In generic terms, the function looks like this:

```
$(nth,item,n1,n2,...)
```

where:

item — a counter; range is 0 to 7.

n — a list of items separated by comma; maximum of eight items in the list.

If *item* exceeds *n*, then Diesel ignores this function.

Here are examples of using the function with numbers and text:

```
$(nth,2,2,3,4,5,6,7) returns 6.7
$(nth,1,Tailoring,proge,CAD) returns proge
```

strlen

The **strlen** function returns the number of characters in the string (short for “string length”). This function is useful for finding the length of a string before applying another function, such as **substr**.

```
$(strlen,Tailoring progeCAD) returns 18
```

If the string is surrounded by quotation marks, Diesel ignores them.

```
$(strlen,"Tailoring progeCAD") also returns 18
```

This function also works with numbers and system variables:

```
$(strlen,3.14159) returns 7
$(strlen,$(getvar,platform)) returns 38
```

substr

The **substr** function returns a portion of a string (short for “sub string”). This is useful for extracting text from a longer portion. Generically, the function looks like this:

```
$(substr,string,start,length)
```

where

string — specifies the text to be handled.

start — indicates the starting position of the substring; first character is #1.

length — specifies the length of the substring; optional. If left out, the entire rest of the string is returned.

Here are some examples of this function at work:

```
$(substr,Tailoring progeCAD,5) returns      oring progeCAD  
$(substr,Tailoring progeCAD,5,7) returns    oring p
```

If the string is surrounded by quotation marks, Diesel ignores them.

```
$(substr,"Tailoring progeCAD",5) also returns oring progeCAD
```

This function also works with numbers and system variables:

```
$(substr,3.14159,1,4) returns      3.14  
$(substr,$(getvar,platform),5,15) returns    osoft Windows N
```

upper

The **upper** function converts text strings to uppercase characters. (There is no “lower” function in Diesel.) It works with text and system variables, as follows:

```
$(upper,"Tailoring progeCAD") returns    TAILORING PROGECAD  
$(upper,$(getvar,platform)) returns      MICROSOFT WINDOWS NT VERSION 5.0 (X86)
```

The function also works with numbers, but leaves them unchanged.

System Functions

The system functions are a collection of miscellaneous functions.

edtime

The **edtime** function formats the display of the system time. Notice that Windows displays the time, such as 11:37 AM, at the right end of the task bar, but you can have Diesel display a customized version of the date and time at the left end of progeCAD’s status bar.

This function reads the date and time from the **Date** system variable, and then formats it according to your instructions. Generically, the function looks like this:

```
$(edtime,$(getvar,date),format)
```

where

format — specifies how the date and time should be displayed, as illustrated by the table below.

When *format* contains text that Diesel cannot interpret, it is displayed literally. The table shows date formatting codes for a date of September 5, 2010:

Date Formats	Meaning	Example
D	Single-digit date	5
DD	Dual-digit date	05
DDD	Three-letter day	Fri
DDDD	Full-letter day	Friday
M	Single-digit month	9
MO	Dual-digit month	09
MON	Three-letter month	Sep
MONTH	Full-letter month	September
YY	Dual-digit year	10
YYYY	Four-digit year	2010

The table below lists time formatting codes for a time of 1:51:23.702AM:

Time Formats	Meaning	Example
H	Single-digit hour	1
HH	Dual-digit hour	01
MM	Minutes	51
SS	Seconds	23
MSEC	Milliseconds	702
AM/PM	Uppercase AM or PM	AM
am/pm	Lowercase AM or PM	am
A/P	Abbreviated uppercase	A
a/p	Abbreviated lowercase	a

TIPS To use commas in the format code, surround them with "," so that Diesel does not read the comma as an argument separator.

The quotation mark trick does not work for words like "Date" and "Month": Diesel returns 1date and 7onth.

The date and time codes are case-insensitive; **D** and **d** work the same. The exceptions are for the **AM/PM** and **am/pm** codes.

When the **AM/PM** and **A/P** format codes are used, Diesel displays the 12-hour clock; when they are left out, Diesel displays the 24-hour clock.

The **AM/PM** and **A/P** format codes must be entered with the slash. If, say, PM is entered, then Diesel returns P literally and reads **M** as the single-digit month code.

Here are some examples of using the **EdTime** function:

```
$(edtime,$(getvar,date),H:MMam/pm)           returns 11:58am
$(edtime,$(getvar,date),DDD"," DD-MO-YY)     returns Thu, 01-07-10
$(edtime,$(getvar,date),DDD"," d mon"," YYYY) returns Thu, 1 Jul, 2010
```

eval

The **eval** function displays text on the status bar:

```
Command: modemacro
Enter new value for MODEMACRO, or . for none <"">: $(eval,"This is text")
Displays This is text on the status bar.
```

It is equivalent to using the **ModeMacro** command without Diesel:

```
Command: modemacro  
Enter new value for MODEMACRO, or . for none <"">: This is text
```

`getvar`

The **getvar** function gets the values of system variables (short for “get variable”).

```
$(getvar,lunits)          returns 4
```

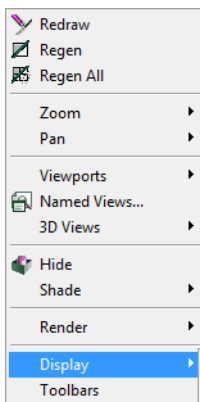
Diesel Programming Tips

Here are some tips for working with Diesel:

- Each argument must be separated by a comma; there must be no spaces within the expression.
- The maximum length of a Diesel macro is 240 characters; the maximum display on the status bar is 32 characters.
- The **ModeMacro** system variable outputs text directly to the status bar until it reaches a **\$(**, and then it begins evaluating the macro.
- To prevent evaluation of a Diesel macro, use quoted strings : "**\$(+,1)**"; to display quotation marks on the status bar, use double quotations: **""Test""**
- Use the **MacroTrace** system variable to debug macros.
- Use AutoLISP’s (**strcat**) function to string together Diesel macros within AutoLISP.
- Use the **\$M=** construct to use Diesel expressions in menu and toolbar macros.

John Walker, the programmer who created Diesel, notes that additional functions could be made available in Diesel, but are not. These unimplemented functions include **setvar** and **time**. He provides instructions for accessing the Diesel source code and recompiling it with other functions. See www.fourmilab.ch/diesel.

Diesel in Menus and Toolbars



Let’s now see how Diesel code can be used in menu and toolbar macros — and discover just how complex Diesel programming can get. For our examples, we will use ones from AutoCAD, because progeCAD doesn’t use Diesel in its default menu files (menus, shortcuts, and so on are hardcoded into progeCAD).

The most common use of Diesel by AutoCAD is to turn check marks on and off in menus; Diesel is also used to determine which drawings states are active, such as grips, model space, and reference editing.

The **View** menu shows several examples: select **Display**, and then notice the check mark in front of **Status Bar** and the names of other user interface elements.

It is trivial to *display* check marks in menus: simply prefix a

word like “Status Bar” with the **!** (exclamation-dot) metacharacter. *Toggling* the display of check marks is trickier in AutoCAD, requiring the use of a Diesel macro that look like this:

```
$(if,$(eq,$(getvar,varname),1),!)&Status Bar
```

(In this part of the chapter, Diesel code is shown in **color**, while menu code is shown in black. The job of the macro above is to determine whether the check mark should be displayed in front of Normal. This macro is called the “name” (or *label* in earlier releases).

Parsing the Name Macro

Let’s figure out the meaning of the followin macro, character by character.

```
$(if,$(eq,$(getvar,attmode),1),!)&Normal
```

To make it easier to read, I’ll parse the code. *Parsing* places each function on its own line, as follows:

<code>\$(if,</code>	<i>If AttMode...</i>
<code> \$(eq,</code>	<i>equals...</i>
<code> \$(getvar,attmode),</code>	<i>(get value of AttMode)</i>
<code>),</code>	<i>1</i>
<code> !)</code>	<i>then display the check mark.</i>
<code>&Normal</code>	<i>And display Normal.</i>

Let’s look at the meaning of the code in greater detail.

`$(if, ... !)`

The **if** function tests the **eq** expression that follows. If the expression is true, then the **!** metacharacter is executed and the check mark is displayed; if not, it’s not.

progeCAD’s Checkmark Function

progeCAD doesn’t use Diesel to toggle checkmarks in menu items, unlike AutoCAD. When you export the menu from progeCAD with the Customize command’s Export button, you will see the following code in Notepad for the View | Display | Status Bar item:

```
[Mnultem-124]
Name=Status &Bar F10
Command=_STATBAR;_T
HelpString=Turns the Status Bar on and off
ChekVar=WNDLSTAT
Visibility=239
SubLevel=2
```

The checkmark is toggle by the ChekVar parameter, which checks the value of system variable WndlStat.

`$(eq, ... 1)`,

The **eq** function tests the value of the AttMode system variable returned by GetVar. If the value is **1**, then the **eq** function returns true; if any other value, it returns false.

`$(getvar,attmode)`,

The **getvar** function retrieves the value of the AttMode system variable. While this system variable has three possible meanings, it has just two possible meanings in this macro:

AttMode	Meaning	Meaning for Diesel Macro
0	Off: all attributes are invisible.	False (0)
1	Normal: visible attributes are displayed.	True (1)
2	On: all attributes are displayed.	False (0)

`&Normal`

The **&Normal** displays the word Normal in the menu, with the letter N underlined. (The & metacharacter appears in front of the letter to be underlined, for keyboard access.)

Note that a second macro does the actual toggling of the attributes' display through the **AttDisp** command:

```
_attdisp _n
```

Parsing Diesel in Macros

In the example above, Diesel was used to control the display of a menu item. Let's now turn to another macro. This one checks conditions before executing commands:

```
$M=$(if,$(eq,$(substr,$(getvar,cmdnames),1,4),GRIP),_move,^C^C_move)
```

Briefly, this macro checks whether grips editing is active. If so, it executes the Move command; if not, it cancels the current command, and then executes the Move command.

When Diesel is used with menu and toolbar macros, it must be prefixed with the **\$M=** metacharacter. (I don't know why "M" is used — for ModeMacro, perhaps?)

The code that checks if grips are enabled before executing the **Move** command:

<code>\$M=</code>	<i>Start Diesel macro.</i>
<code>\$(if,</code>	<i>If the value of CmdNames...</i>
<code>\$(eq,</code>	<i>equals</i>
<code>\$(substr,</code>	<i>the substring</i>
<code>\$(getvar,cmdnames)</code>	<i>(gotten from the system variable CmdNames)</i>
<code>,1,4)</code>	<i>first four characters</i>
<code>,GRIP)</code>	<i>"GRIP"</i>
<code>_move,</code>	<i>Then execute the Move command.</i>
<code>^C^C_move)</code>	<i>Otherwise, cancel the current command (^C^C), and then execute the Move command.</i>

In other words: if the four characters of the system variable CmdNames equal "GRIP" then execute the Move command; if not, cancel the current command and then execute the **Move** command.

Bitcode Macros

Not all system variables are straightforward toggles, where progeCAD just checks if the value is 0 or 1. Some are *bitcodes*, where three or more integers can represent many values. Bitcodes are used when different combinations of settings are possible. The most extreme is the OsnapMode system variables, with its 15 bitcodes representing ENDpoint, INTersection, and other object snap modes. When dealing with bitcodes, you need to use Diesel's **and** operator, instead of **eq**.

An example is the macro that toggles the display of the check mark in AutoCAD's **View | Display | UCS Icon** menu.

```
$(if,$(and,$(getvar,ucsicon),1),1)&On
```

UscIcon	BitCode	Meaning
0	0	UCS icon not displayed.
1	1	UCS icon is displayed at the lower-left corner of the current viewport.
2	2	UCS icon is displayed at the origin, if possible.
3	1+2	UCS icon is on.

The longest Diesel macro may well be this one:

```
^C^C_dview$M=$(if,$(or,$(eq,$(getvar,tilemode),1),$(!,$(getvar,cvport),1)),$(if,$(and,$(getvar,viewmode),2),$(if,$(and,$(getvar,viewmode),4),_all _cl _off _cl _b _on^M, _all _cl _off^M), _all _cl _f $(getvar,frontz)^M)^Z)
```

Diesel in AutoLISP

There are two ways to use Diesel expressions inside AutoLISP routines: with the **setvar** function, and the **menucmd** function. I don't know if there is a preference for either among the programming community; either way, Diesel is accessed in an indirect manner.

Via the Setvar Function

AutoLISP's **setvar** function is used in conjunction with the ModeMacro system variable. You'll recall from earlier tutorials that the ModeMacro system variable executes Diesel from the 'Command:' prompt. The same trick is used here.

To show how this works, I'll write an AutoLISP routine to display the fillet radius on the status bar — using Diesel (shown in **color**).

```
(defun frad ()  
  (setvar "modemacro" "Current fillet radius: $(getvar,filletrad)")  
)
```

Recall that the **FilletRad** system variable contains the current setting for the filleting radius.

Concatenate Two Diesel Strings

To display more than one piece of information on the status bar, I use AutoLISP's **strcat** function to concatenate the two Diesel strings to the **ModeMacro** system variable in one piece. The following AutoLISP code displays the two chamfer distances at the status bar:

```
(defun chab ()
  (setvar "modemacro"
    (strcat "Chamfer A: $(getvar,chamfera)" "Chamfer B: $(getvar,chamferb)"))
  )
)
```

Via the MenuCmd Function

The second method for using Diesel in AutoLISP functions employs the **menucmd** function along with the **M=** construct, as follows:

```
(defun chab ()
  (menucmd "M=Current fillet radius: $(getvar,filletrad)")
  )
)
```

The **M=** should be familiar from the earlier discussion of using Diesel inside of menu macros.

Understanding DXF

dWG and DXF are the two most important file formats in the Province of AutoCAD, and maybe even in the Country of CAD. There are one billion, two billion, three billion, or more CAD drawings in DWG format — all rough estimates, depending on the source you read.

Despite Autodesk calling DWG “a standard,” it does not document the file format, because the company wants to be able to make changes to it without being beholden to a standards body. Because Autodesk keeps DWG closed, the Open Design Alliance was formed to document the format, and to provide programming code that allows other CAD companies to more easily access drawings saved in DWG.

There is a public face to DWG, and it is DXF (short for “drawing interchange format”). Autodesk created DXF in the earliest days of AutoCAD as *the* means by which third-party developers could access the data stored in drawing files.

This chapter describes the DXF format, which is also read and written by progeCAD and many other graphical software packages.

References

On the Internet, you can read Autodesk’s DXF references at www.autodesk.com/dxf for AutoCAD Release 13 through to AutoCAD 2009.

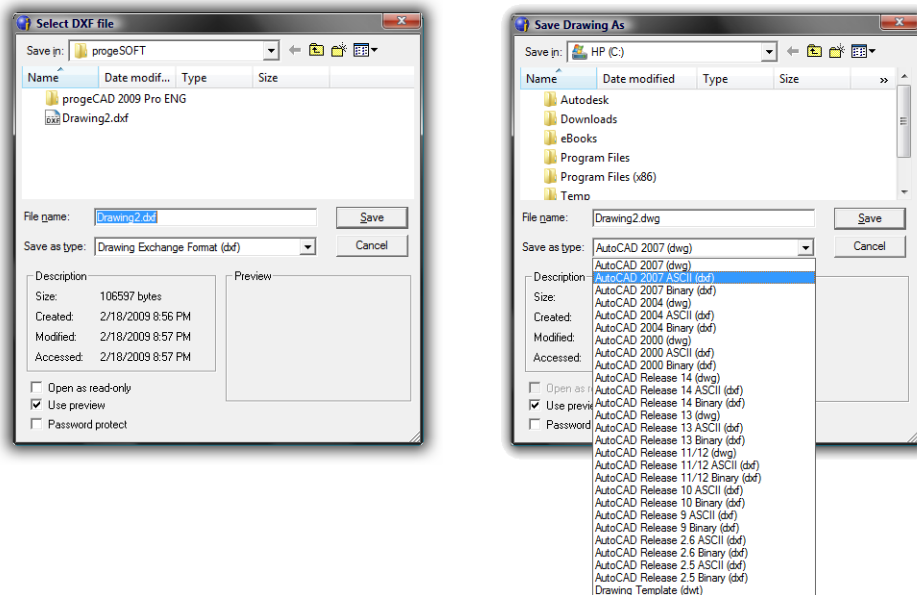
Autodesk does not document DWG, but the Open Design Alliance does. Its R13-2007 DWG specification is in RTF (rich text format) at www.opendesign.com/guestfiles.

In This Chapter

- DWG and DXF file format versions.
- The format of DXF.
- Object properties.
- Group codes.

Introduction

DXF files can be created from progeCAD drawings using by the **DxfOut** command (short for “dxf output”).



*Left: DxfOut dialog box.
Right: SaveAs dialog box.*

As an alternative, you can use the **SaveAs** command, and then select one of the many “AutoCAD DXF” formats from the **Files of type** droplist.

DXF Formats

progeCAD outputs DXF in two primary formats:

- **ASCII** is a human-readable format, and can be opened with any text editor, such as Notepad. When you select ASCII format, you can also specify the number of decimal places; some CAM software, for example, requires that DXF format use four decimal places only.
- **Binary** is a compact format that can be written and read more quickly by computer software. Be aware that not all software can read the binary format of DXF.

In addition, you can specify the release with which to be compatible:

- AutoCAD 2007 (also compatible with AutoCAD 2008 and 2009)
- AutoCAD 2004 (also compatible with AutoCAD 2005 and 2006)
- AutoCAD 2000 (also compatible with AutoCAD 2000i and 2002)
- Release 14
- Release 13
- Release 11/12
- Release 10

- Release 9
- Version 2.6
- Version 2.5

The DXF format is also used by LISP to access entity data through *dotted pairs*. In the DXF documentation later in this chapter, “**DXF**” indicates group codes used by *.dxf* files only, and “**APP**” indicates group code used in LISP only. All other group codes apply to both *.dxf* files and dotted pairs.

DWG and DXF Content

The DWG and DXF formats share similarities in their file structure.

Version Identifier. Both start off with a version identifier. For instance, files saved in AutoCAD 2005 format show an identifier of **AC1018**. (AC = AutoCAD; 1018 = the 18th revision of the file format.)

Table Sections. Next, both formats have *table* sections. These are items common to the entire drawing, such as system variables, layers, and blocks.

Entity Sections. Then comes the part you’re waiting for: entities and their properties. Lines, circles, polylines, and other geometric objects are called entities.

Object Sections. In the DXF documentation, the word *object* refers to non-graphical things, such as layers, layouts, and multiline styles. The object section holds everything that isn’t an entity or defined by the symbol tables.

Miscellaneous Sections. Near the end of both files are miscellaneous sections, such as the bitmap used to display preview images.

End of File. Finally, both kinds of files end with the **EOF** marker, which indicates the end of the file.

In summary, DXF and DWF files consists of the following sections:

Section	Comment
HEADER	Version number and system variables.
CLASSES	Application-defined classes in the BLOCKS, ENTITIES, and OBJECTS sections.
TABLES	Contains these symbol tables:
APPID	Application identification.
BLOCK_RECORD	Block references.
DIMSTYLE	Dimension styles.
LAYER	Layer names and settings.
LTYPE	Linetype names and definitions.
STYLE	Text styles.
UCS	Saved user coordinate systems.
VIEW	Named views.
VPORT	Viewport configurations.
BLOCKS	Block definitions.
ENTITIES	Graphical objects, including “inserts” (block references).
OBJECTS	Nongraphical objects in the drawing.
THUMBNAILIMAGE	Preview image (optional).
EOF	End of file.

Miscellaneous Comments

DWG is always binary; DXF can be either ASCII or binary.

Group codes can be in any order. A “0” (zero) indicates the end of a section.

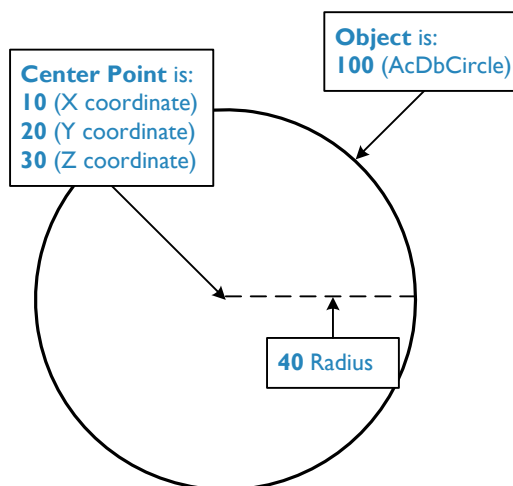
The **Select Objects** option of the DxfOut (or SaveAs) command creates DXF files with only the ENTITIES section.

DXF Format

The format of DXF is in *pairs of data*. The first part of the data explains the meaning of the second. The first part is called the “group code”; the second is its “associated value.”

As an example, here is the DXF data relating to a circle, which is defined by its center point and radius:

DXF Data	Comment
100 AcDbCircle	Entity code... ...is a circle.
10 6.858256073433437	Center point... ... is the x coordinate.
20 4.147281918113382	Center point... ...is the y coordinate.
30 0.0	Center point... ...is the z coordinate.
40 1.722175158117808	Radius... ...is a double-precision real number.
0 ENDSEC	End of section... ...is the end.



Header Section of DXF Files

9		40	0
\$ACADVER	\$FILLMODE	0.18	9
1	70	9	\$DIMSE2
AC1018	1	\$DIMEXO	70
9	9	40	0
\$ACADMANTVER	\$QTEXTMODE	0.0625	9
70	70	9	\$DIMTAD
39	0	\$DIMDLI	70
9	9	40	0
\$DWGCODEPAGE	\$MIRRTEXT	0.38	9
3	70	9	\$DIMZIN
ANSI_1252	0	\$DIMRND	70
9	9	40	0
\$INSBASE	\$LTSCALE	0.0	9
10	40	9	\$DIMBLK
0.0	1.0	\$DIMDLE	1
20	9	40	9
0.0	\$ATTMODE	0.0	\$DIMASO
30	70	9	70
0.0	1	\$DIMEXE	1
9	9	0.18	9
\$EXTMIN	\$TEXTSIZE	9	\$DIMSHO
10	40	\$DIMTP	70
5.136080915315629	0.2	40	1
20	9	0.0	9
2.425106759995574	\$TRACEWID	9	\$DIMPOST
30	40	\$DIMTM	1
0.0	0.05	40	9
9	9	0.0	\$DIMAPOST
\$EXTMAX	\$TEXTSTYLE	9	1
10	7	\$DIMTXT	40
8.580431231551244	Standard	0.18	9
20	9	9	\$DIMALT
5.86945707623119	\$CLAYER	\$DIMCEN	70
30	8	40	0
0.0	0	0.09	9
9	9	9	\$DIMALTD
\$LIMMIN	\$CELTYPE	\$DIMTSZ	70
10	6	40	2
0.0	ByLayer	0.0	9
20	9	9	\$DIMALTF
0.0	\$CECOLOR	\$DIMTOL	40
9	62	70	25.4
\$LIMMAX	256	0	9
10	9	9	\$DIMLFAC
12.0	\$CELTSCALE	\$DIMLIM	40
20	40	70	1.0
9.0	1.0	0	9
9	9	9	\$DIMTOFL
\$ORTHOMODE	\$DISPSILH	\$DIMTIH	70
70	70	70	0
0	0	1	9
9	9	9	\$DIMTVP
\$REGENMODE	\$DIMSCALE	\$DIMTOH	40
70	40	70	0.0
1	1.0	1	9
9	9	9	\$DIMTIX
	\$DIMASZ	\$DIMSE1	70
		70	

0	0	0	0.0
9	9	9	9
\$DIMSOXD	\$DIMALTZ	\$DIMLDRBLK	\$LIMCHECK
70	70	1	70
0	0		0
9	9	9	9
\$DIMSAH	\$DIMALTZT	\$DIMLUNIT	\$SCHAMFERA
70	70	70	40
0	0	2	0.0
9	9	9	9
\$DIMBLK1	\$DIMUPT	\$DIMLWD	\$SCHAMFERB
1	70	70	40
	0	-2	0.0
9	9	9	9
\$DIMBLK2	\$DIMDEC	\$DIMLWE	\$SCHAMFERC
1	70	70	40
	4	-2	0.0
9	9	9	9
\$DIMSTYLE	\$DIMTDEC	\$DIMTMOVE	\$SCHAMFERD
2	70	70	40
Standard	4	0	0.0
9	9	9	9
\$DIMCLRD	\$DIMALTU	\$LUNITS	\$SKPOLY
70	70	70	70
0	2	2	0
9	9	9	9
\$DIMCLRE	\$DIMALTTD	\$LUPREC	\$TDCREATE
70	70	70	40
0	2	4	2453116.603633889
9	9	9	9
\$DIMCLRT	\$DIMTXSTY	\$SKETCHINC	\$TDUCREATE
70	7	40	40
0	Standard	0.1	2453116.895300555
9	9	9	9
\$DIMTFAC	\$DIMAUNIT	\$FILLETRAD	\$TDUPDATE
40	70	40	40
1.0	0	0.0	2453116.606109838
9	9	9	9
\$DIMGAP	\$DIMADEC	\$AUNITS	\$TDUUPDATE
40	70	70	40
0.09	0	0	2453116.897776505
9	9	9	9
\$DIMJUST	\$DIMALTRND	\$AUPREC	\$TDINDWG
70	40	70	40
0	0.0	0	0.0020187731
9	9	9	9
\$DIMSD1	\$DIMAZIN	\$MENU	\$TDUSRTIMER
70	70	1	40
0	0	.	0.0020180556
9	9	9	9
\$DIMSD2	\$DIMDSEP	\$ELEVATION	\$USRTIMER
70	70	40	70
0	46	0.0	1
9	9	9	9
\$DIMTOLJ	\$DIMATFIT	\$PELEVATION	\$ANGBASE
70	70	40	50
1	3	0.0	0.0
9	9	9	9
\$DIMTZIN	\$DIMFRAC	\$THICKNESS	\$ANGDIR
70	70	40	70

0	0.0	\$UCSORGFRONT	0.0
9	30	10	30
\$PDMODE	0.0	0.0	0.0
70	9	20	9
0	\$UCSXDIR	0.0	\$PUCSORGBOTTOM
9	10	30	10
\$PDSIZE	1.0	0.0	0.0
40	20	9	20
0.0	0.0	\$UCSORGBACK	0.0
9	30	10	30
\$PLINEWID	0.0	0.0	0.0
40	9	20	9
0.0	\$UCSYDIR	0.0	\$PUCSORGLEFT
9	10	30	10
\$SPLFRAME	0.0	0.0	0.0
70	20	9	20
0	1.0	\$PUCSBASE	0.0
9	30	2	30
\$SPLINETYPE	0.0	0.0	0.0
70	9	9	9
6	\$UCSORTHOREF	\$PUCSNAME	\$PUCSORGRIGHT
9	2	2	10
\$SPLINESEGS	9	9	0.0
70	\$UCSORTHOVIEW	\$PUCSORG	20
8	70	10	0.0
9	0	0.0	30
\$HANDSEED	9	20	0.0
5	\$UCSORGTOP	0.0	9
30	10	30	\$PUCSORGFRONT
9	0.0	0.0	10
\$SURFTAB1	20	9	0.0
70	0.0	\$PUCSXDIR	20
6	30	10	0.0
9	0.0	1.0	30
\$SURFTAB2	9	20	0.0
70	\$UCSORGBOTTOM	0.0	9
6	10	30	\$PUCSORGBACK
9	0.0	0.0	10
\$SURFTYPE	20	9	0.0
70	0.0	\$PUCSYDIR	20
6	30	10	0.0
9	0.0	0.0	30
\$SURFU	9	20	0.0
70	\$UCSORGLEFT	1.0	9
6	10	30	\$USER11
9	0.0	0.0	70
\$SURFV	20	9	0
70	0.0	\$PUCSORTHOREF	9
6	30	2	\$USER12
9	0.0	9	70
\$UCSBASE	9	\$PUCSORTHOVIEW	0
2	\$UCSORGRIGHT	70	9
9	10	0	\$USER13
\$UCSNAME	0.0	9	70
2	20	\$PUCSORGTOP	0
\$UCSORG	0.0	10	9
10	30	0.0	\$USER14
0.0	0.0	20	70
20	9		0
			9

\$USER15	1.000000000000000E+20	\$MEASUREMENT	0.0
70	30	70	9
0	1.000000000000000E+20	0	\$OLESTARTUP
9	9	9	290
\$USERR1	\$PEXTMAX	\$CELWEIGHT	0
40	10	370	9
0.0	-1.000000000000000E+20	-1	\$SORTENTS
9	20	9	280
\$	-1.000000000000000E+20	\$ENDCAPS	127
40	30	280	9
0.0	-1.000000000000000E+20	0	\$INDEXCTL
9	9	9	280
\$USERR3	\$PLIMMIN	\$JOINSTYLE	0
40	10	280	9
0.0	0.0	0	\$HIDETEXT
9	20	9	280
\$USERR4	0.0	\$LWDISPLAY	1
40	9	290	9
0.0	\$PLIMMAX	0	\$XCLIPFRAME
9	10	9	290
\$USERR5	12.0	\$INSUNITS	0
40	20	70	9
0.0	9.0	1	\$HALOGAP
9	9	9	280
\$WORLDVIEW	\$UNITMODE	\$HYPERLINKBASE	0
70	70	1	9
1	0	9	\$OBSCOLOR
9	9	9	70
\$SHADEDGE	\$VISRETAIN	\$STYLESHEET	257
70	70	1	9
3	1	9	\$OBSLTYPE
9	9	9	280
\$SHADEDIF	\$PLINEGEN	\$XEDIT	0
70	70	290	9
70	0	1	\$INTERSECTIONDISPLAY
9	9	9	280
\$TILEMODE	\$PSLTSCALE	\$CEPSNTYPE	0
70	70	380	9
1	1	0	\$INTERSECTIONCOLOR
9	9	9	70
\$MAXACTVP	\$TREEDEPTH	\$PSTYLEMODE	257
70	70	290	9
64	3020	1	\$DIMASSOC
9	9	9	280
\$PINSBASE	\$CMLSTYLE	\$FINGERPRINTGUID	2
10	2	2	9
0.0	Standard	{35221D38-97D8-4074-9FA9-AC5AB89E4A0E}	\$PROJECTNAME
20	9	9	1
0.0	\$CMLJUST	\$VERSIONGUID	0
30	70	2	ENDSEC
0.0	0	{855B33C7-1572-44F4-A59A-EF5E25C06767}	
9	9	9	
\$PLIMCHECK	\$CMLSCALE	\$EXTNAMES	
70	40	290	
0	1.0	1	
9	9	9	
\$PEXTMIN	\$PROXYGRAPHICS	\$PSVPSCALE	
10	70	40	
1.000000000000000E+20	1		
20	9		

Notice the pairings: group code **100** alerts you that the associated value is the name of an entity. The entity is a circle, in this case.

The group code **10** alerts you to the x coordinate of the circle's center (6.858256073433437). Similarly, group code **20** is followed by the y coordinate, and group code **30** reports the z coordinate.

Group code **40** is followed by the radius (1.722175158117808). The definition of the circle ends with group code **0** reporting the end of the section.

Object Properties

The group codes I unpacked for you above relate to the circle's geometry. But circles also have *properties*, such as color, layer, and linetype. A similar group code system describes properties. For example, group code **6** describes the linetype, group code **8** the layer name, and **62** the color.

Properties	Comment
6 ByLayer	Linetype... ...is ByLayer.
8 0	Layer name... ...is 0.
62 1	Color... ...is 1 (red).

Properties can apply to the entire drawing or to a layer (global), or to individual entities (local). Thus, you find the linetype, layer, color, and other properties more than once in DXF files.

For example, a circle is on layer. When the circle's color is BYLAYER, it is the same color as assigned to the layer; but the color of the circle can be overridden locally.

Group code **9** is only used in the header section of the DXF file; its purpose is to announce the start of the next section.

Group Codes

Understanding group codes is crucial to understanding DXF. But they can be tricky to understand, because their meaning changes subtly with the entity type.

For circles, lines, and other entities, 10 always refers to the x coordinate. But the x coordinate's reference differs: for some, it is at the center point, for others it is at the end point. For example, you saw that group code 10 reports the x coordinate of a circle's center point. For lines, however, group code 10 reports the x coordinate of the first endpoint. The same meaning, but slightly different interpretation.

Autodesk groups the group codes into categories, which helps deciphering DXF files. For example, all group codes in the range from 10 to 39 are double-precision 3D points (x, y, or z). Group codes in the range from 0 to 9 are strings. And so on. There are even some negative group codes, which are used by application data.

TIPS When writing software to read DXF file, the software needs to read two lines of data at a time. First, the program reads one line to get the group code. Knowing the group code, the program reads the second line to find the value of the group code.

The longest line in a DXF file can be 256 characters; additional characters are truncated.

HEADER Section

DXF files start with the Header section. The start of the Header section is:

```
0
SECTION
2
HEADER
```

And the end is signaled by:

```
0
ENDSEC
```

A typical entry looks like this:

```
9
$ACADVER
1
AC1018
```

The meaning of the code is as follows.

Header	Comments
9 \$ACADVER	Name of the system variable is... ...AcadVer (DXF version number).
1 AC1018	Value of the system variable is... ...AC1018

Group code **9** is found only in the header section of the DXF file; its purpose is to announce the impending name of a system variable, or other header data.

All data in the Header section follows this pattern:

```
9
$variableName
groupCodeNumber
value
```

The group code that follows reports the value of the system variable. In the \$AcadVer example on the previous page, it is **1**, because it is a piece of text. If the value were an integer, then group code would be **70**; if a real number, the group code is **40**; if x,y,z coordinates, then **10(x)**, **20(y)**, and **30(z)**. And the occasional sysvar uses a special group code, such as **290** for PlotStyleMode.

The Header contains the settings for system variables. But not all sysvars, because the value of some are not stored (their values are dynamic, such as the current time and the name of the current drawing). Others sysvars are stored in the Windows registry, because they are the same for all drawings, such as the user login name.

Version Numbers

Autodesk uses these number to distinguish between drawing database revisions:

DXF Version	AutoCAD Release
AC1000	V1.0
AC1001	V1.2
AC1002	V1.4
AC1003	V2.5
AC1004	V2.6
AC1005	R9
AC1006	R10
AC1009	R11 and R12
AC1012	R13
AC1014	R14
AC1015	2000
AC1016	2002
AC2017	2004
AC1018	2005
AC1019	2006
AC1020	2007
AC1021	2008
AC1022	2009
AC1023	2010

CLASSES Section

The Classes section contains *application-defined* classes that are used later by the BLOCKS, ENTITIES, and OBJECTS sections.

The start of the Classes section is:

```
0  
SECTION  
2  
CLASSES
```

And the end is signaled by:

```
0  
ENDSEC
```

In between are some of the following classes. (Applications can add their own classes to this list.)

Classes

```
ACDBDICTIONARYWDFLT  
ACDBPLACEHOLDER  
ARCALIGNEDTEXT  
DICTIONARYVAR  
HATCH  
IDBUFFER  
IMAGE  
IMAGEDEF  
IMAGEDEF_REACTOR  
LAYER_INDEX
```

LAYOUT
 LWPOLYLINE
 OBJECT_PTR
 OLE2FRAME
 PLOTSETTINGS
 RASTERVARIABLES
 RTEXT
 SORTENTSTABLE
 SPATIAL_INDEX
 SPATIAL_FILTER
 WIPEOUT
 WIPEOUTVARIABLES

All data in the Classes section follows this pattern:

0		
CLASS		Record type (CLASS). Identifies beginning of a CLASS record
1		
<i>classDxfRecord</i>		Class DXF record name; always unique
2		
<i>className</i>		Class DXF record name; always unique
3		
<i>appName</i>		Application name.
90		
<i>flag</i>		Bit-code to indicate the capabilities of this object when displayed as a proxy:
		0 = No operations allowed
		1 = Erase allowed
		2 = Transform allowed
		4 = Color change allowed
		8 = Layer change allowed
		16 = Linetype change allowed
		32 = Linetype scale change allowed
		64 = Visibility change allowed
		127 = All operations except cloning allowed
		128 = Cloning allowed
		255 = All operations allowed
		32768 = R13 format proxy
280		
<i>flag</i>		1 = Class not loaded when this DXF file created
		0 = Otherwise
281		
<i>flag</i>		1 = Class derived from AcDbEntity class; can be in BLOCKS or ENTITIES section.
		0 = May appear in the OBJECTS section only.

TABLES Section

Drawings consist of objects and *tables*. Tables describe the global aspects of drawings, such as its layers, text styles, and blocks. The start of the Table section is:

0
 SECTION
 2
 TABLES

And the end is signaled by:

0
 ENDSEC

The Tables section contains these symbol tables:

Tables	Comment
APPID	Application identification.
BLOCK_RECORD	Block references.
DIMSTYLE	Dimension styles.
LAYER	Layer names and settings.
LTYPE	Linetype names and definitions.
STYLE	Text styles.
UCS	Saved user coordinate systems.
VIEW	Named views.
VPORT	Viewport configurations.

All data in the Tables section follow this pattern:

```
0
TABLE
2

```

Common table group codes, repeat for each entry

```
0

```

BLOCKS Section

The Blocks section holds block definitions. A block definition consists of these parts:

- The entities that make up the block.
- Basepoint (the point at which the block is inserted).
- Name of the block.

(The Entities section, later in the DXF file, defines where and how the blocks are inserted in the drawing.)

The start of the Blocks section is:

```
0
SECTION
2
BLOCKS
```


And the end is signaled by:

```
0  
ENDSEC
```

The block definition generically looks like this:

```
0  
BLOCK  
5  
handleNumber  
100  
AcDbEntity  
8  
layerName  
100  
AcDbBlockBegin  
2  
blockName  
70  
flag  
10  
xCoordinate  
20  
yCoordinate  
30  
zCoordinate  
3  
blockName  
1  
xrefPath
```

Blocks consist of one or more entities. To define the start of an entity:

```
0  
entityType  
dataAboutTheEntity
```

See the Entities section for details on defining entities. The block definition ends with:

```
0  
ENDBLK  
5  
handle  
100  
AcDbBlockEnd
```

ENTITIES Section

The Entities section holds graphical objects, including “inserts” (block references).

The start of the Entities section is:

```
0  
SECTION  
2  
ENTITIES
```

And the end is signaled by:

```
0  
ENDSEC
```

The generic entry looks like this:

```
0  
entityType  
5  
handle  
330  
pointerToOwner  
100  
AcDbEntity  
8  
layerName  
100  
AcDbclassName  
dataSpecificToEntity
```

Entities are:

Entity	Comment
3DFACE	3D faces (surfaces)
3DSOLID	3D solids created by ShapeManager
ACAD_PROXY_ENTITY	Proxy or zombie objects created by other applications
ARC	Arcs
ATTDEF	Attribute definitions
ATTRIB	Attributes values
BODY	3D solids created by ShapeManager
CIRCLE	Circles
DIMENSION	Associative dimensions
ELLIPSE	True ellipses
HATCH	Associative hatch patterns
IMAGE	Raster images
INSERT	Inserted blocks
LEADER	Leaders
LINE	Lines
LWPOLYLINE	Lightweight polylines
MLINE	Multilines
MTEXT	Multiline text
OLEFRAME	Object linking and embedding objects
OLE2FRAME	OLE version 2 objects
POINT	Points
POLYLINE	Polylines
RAY	Rays (semi-infinite lines)
REGION	2D regions
SEQEND	End of a polyline sequence
SHAPE	Shapes defined by .shx files
SOLID	2D solid-filled areas
SPLINE	Splines
TABLE	Tables
TEXT	Single-line text
TOLERANCE	Tolerances

VERTEX	Polyline vertices
VIEWPOINT	Viewpoints
WIPEOUT	Wipeout areas
XLINE	Xlines (infinite lines)

OBJECTS Section

The Objects section describes nongraphical objects in the drawing.

The start of the Objects section is:

```
0
SECTION
2
OBJECTS
```

And the end is signaled by:

```
0
ENDSEC
```

A typical entry looks like this:

Object	Comments
ACAD_PROXY_OBJECT	
ACDBDICTIONARYWDFLT	
ACDBPLACEHOLDER	
DICTIONARY	
DICTIONARYVAR	
DIMASSOC	
GROUP	
IDBUFFER	
IMAGEDEF	
IMAGEDEF_REACTOR	
LAYER_INDEX	
LAYER_FILTER	
LAYOUT	Data related to layouts.
MLINESTYLE	Data describing multiline styles.
OBJECT_PTR	Data related to ASE (AutoCAD SQL Extension).
PLOTSETTINGS	Settings used for plotting.
RASTERVARIABLES	Data related to raster images.
SPATIAL_INDEX	Contains no data; can be ignored.
SPATIAL_FILTER	Data related to clipped external references.
SORTENTSTABLE	Specifies the order in which to draw entities; used by DrawOrder.
TABLESTYLE	Data describing table styles.
VBA_PROJECT	Data related to Visual Basic for Applications.
WIPEOUTVARIABLES	Data related to wipeout entities.
XRECORD	Xrecords contain arbitrary data.

THUMBNAILIMAGE Section

The Thumbnail section contains a preview image (in BMP format) of the last-saved drawing view. This section is optional, and exists only if the preview has been saved.

The start of the Thumbnailimage section is:

```
0
SECTION
2
THUMBNAILIMAGE
```

And the end is signaled by:

```
0
ENDSEC
```

A typical entry looks like this:

Thumbnail	Comments
90	Total number of bytes in the image.
310	Preview image data, 256 characters per line.

EOF

The EOF signals the end of the DXF file. (EOF = End of file.)

9

Reproduced from Autodesk's DXF documentation:

Code	Comment
-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. The name changes each time a drawing is opened; it's never saved.
0	
0	Text string indicating the entity type (fixed).
1	Primary text value for an entity.
2	Name (attribute tag, block name, etc) ¹ .
3-4	Other text 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.
39	Entity's thickness if nonzero (fixed).
40-48	Double-precision floating-point values (text height, scale factors, and so on).
48	Linetype scale; double precision floating point scalar value; default value is defined for all entity types.
49	Repeated double-precision 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 ObjectARX 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

- 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. The subclass data marker segregates 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 ObjectARX ²
- 102 Control string, followed by “{<arbitrary name>” 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 control string allowed is “}” as a group terminator. AutoCAD does not interpret these strings except during drawing audit operations. They are for application use ²
- 105 Object handle for DIMVAR symbol table entry.
- 110 UCS origin (appears only if code 72 is set to 1).
DXF: X value; APP: 3D point.
- 111 UCS X-axis (appears only if code 72 is set to 1).
DXF: X value; APP: 3D vector.
- 112 UCS Y-axis (appears only if code 72 is set to 1).
DXF: X value; APP: 3D vector.
- 120–122 DXF: Y value of UCS origin, UCS X-axis, and UCS Y-axis.
- 130–132 DXF: Z value of UCS origin, UCS X-axis, and UCS Y-axis.
- 140–149 Double-precision floating-point values (e.g. points, elevation, and DIMSTYLE settings).
- 170–179 16-bit integer values, such as flag bits representing DIMSTYLE settings.

200

- 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.
- 270–279 16-bit integer values.
- 280–289 16-bit integer values.
- 290–299 Boolean flag value.

300

- 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.
- 370–379 Lineweight enum value (AcDb::LineWeight). Stored and moved around as a 16-bit integer. Custom non-entity objects may use the full range, but entity classes only use 371–379 DXF group codes in their representation, because AutoCAD and AutoLISP both always assume a 370 group code is the entity’s lineweight. This allows 370 to behave like other “common” entity fields.
- 380–389 PlotStyleName type enum (AcDb::PlotStyleNameType).
Stored and moved around as a 16-bit integer. Custom non-entity objects may use the full range, but entity classes only use 381–389 DXF group codes in their representation, for the same reason as the Lineweight range above.

390–399 String representing handle value of the PlotStyleName object, basically a hard pointer, but has a different range to make backward compatibility easier to deal with. Stored and moved around as an object ID (a handle in DXF files) and a special type in AutoLISP. Custom non-entity objects may use the full range, but entity classes only use 391–399 DXF group codes in their representation, for the same reason as the lineweight range above.

400

400–409 16-bit integers.
410–419 String.
420–427 32-bit integer value. When used with True Color; a 32-bit integer representing a 24-bit color value. The high-order byte (8 bits) is 0, the low-order byte an unsigned char holding the Blue value (0–255), then the Green value, and the next-to-high order byte is the Red Value. Converting this integer value to hexadecimal yields the following bit mask: 0x00RRGGBB. For example, a true color with Red==200, Green==100 and Blue==50 is 0x00C86432, and in DXF, in decimal, 13132850.
430–437 String; when used for True Color; a string representing the name of the color.
440–447 32-bit integer value. When used for True Color, the transparency value.
450–459 Long.
460–469 Double-precision floating-point value.
470–479 String.

999+

999 DXF: The 999 group code indicates that the line following it is a comment string. SAVEAS does not include such groups in a DXF output file, but OPEN 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 double-precision 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.

Notes

¹With the introduction of extended symbol names in AutoCAD 2000, the 255-character limit was lifted. There is no explicit limit to the number of bytes per line, although most lines should fall within 2049 bytes.

²255-character maximum; less for Unicode strings.