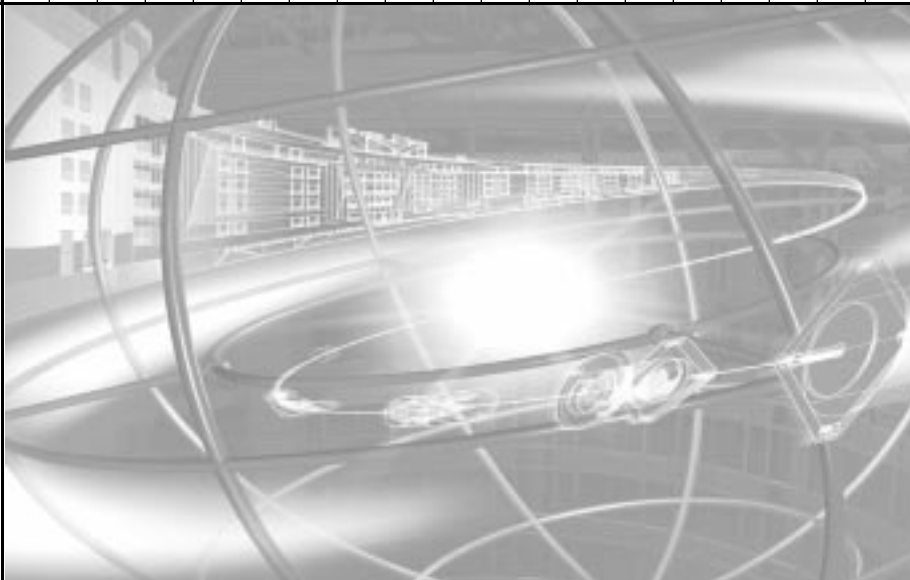


# AutoCAD<sup>®</sup> 2000



## VISUAL LISP<sup>™</sup> DEVELOPER'S GUIDE

# Copyright © 1999 Autodesk, Inc.

All Rights Reserved

**AUTODESK, INC. MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, REGARDING THESE MATERIALS AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS-IS" BASIS.**

**IN NO EVENT SHALL AUTODESK, INC. BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF PURCHASE OR USE OF THESE MATERIALS. THE SOLE AND EXCLUSIVE LIABILITY TO AUTODESK, INC., REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE MATERIALS DESCRIBED HEREIN.**

Autodesk, Inc. reserves the right to revise and improve its products as it sees fit. This publication describes the state of this product at the time of its publication, and may not reflect the product at all times in the future.

## Autodesk Trademarks

The following are registered trademarks of Autodesk, Inc., in the USA and/or other countries: 3D Plan, 3D Props, 3D Studio, 3D Studio MAX, 3D Studio VIZ, 3D Surfer, ADE, ADI, Advanced Modeling Extension, AEC Authority (logo), AEC-X, AME, Animator Pro, Animator Studio, ATC, AUGI, AutoCAD, AutoCAD Data Extension, AutoCAD Development System, AutoCAD LT, AutoCAD Map, Autodesk, Autodesk Animator, Autodesk (logo), Autodesk MapGuide, Autodesk University, Autodesk View, Autodesk WalkThrough, Autodesk World, AutoLISP, AutoShade, AutoSketch, AutoSolid, AutoSurf, AutoVision, Biped, bringing information down to earth, CAD Overlay, Character Studio, Design Companion, Drafrix, Education by Design, Generic, Generic 3D Drafting, Generic CADD, Generic Software, Geodyssey, Heidi, HOOPS, Hyperwire, Inside Track, Kinetix, MaterialSpec, Mechanical Desktop, Multimedia Explorer, NAAUG, Office Series, Opus, PeopleTracker, Physique, Planix, Rastation, Softdesk, Softdesk (logo), Solution 3000, Tech Talk, Texture Universe, The AEC Authority, The Auto Architect, TinkerTech, WHIP!, WHIP! (logo), Woodbourne, WorkCenter, and World-Creating Toolkit.

The following are trademarks of Autodesk, Inc., in the USA and/or other countries: 3D on the PC, ACAD, ActiveShapes, Actrix, Advanced User Interface, AEC Office, AME Link, Animation Partner, Animation Player, Animation Pro Player, A Studio in Every Computer, ATLAST, Auto-Architect, AutoCAD Architectural Desktop, AutoCAD Architectural Desktop Learning Assistance, AutoCAD DesignCenter, Learning Assistance, AutoCAD LT Learning Assistance, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, AutoCDM, Autodesk Animator Clips, Autodesk Animator Theatre, Autodesk Device Interface, Autodesk PhotoEDIT, Autodesk Software Developer's Kit, Autodesk View DwgX, AutoEDM, AutoFlix, AutoLathe, AutoSnap, AutoTrack, Built with ObjectARX (logo), ClearScale, Concept Studio, Content Explorer, cornerStone Toolkit, Dancing Baby (image), Design Your World, Design Your World (logo), Designer's Toolkit, DWG Linking, DWG Unplugged, DXF, Exegis, FLI, FLIC, GDX Driver, Generic 3D, Heads-up Design, Home Series, Kinetix (logo), MAX DWG, ObjectARX, ObjectDBX, Ooga-Chaka, Photo Landscape, Photoscape, Plugs and Sockets, PolarSnap, Powered with Autodesk Technology, Powered with Autodesk Technology (logo), ProConnect, ProjectPoint, Pro Landscape, QuickCAD, RadioRay, SchoolBox, SketchTools, Suddenly Everything Clicks, Supportdesk, The Dancing Baby, Transforms Ideas Into Reality, Visual LISP, and Volo.

## Third Party Trademarks

Élan License Manager is a trademark of Élan Computer Group, Inc.

Microsoft, Visual Basic, Visual C++, and Windows are registered trademarks and Visual FoxPro and the Microsoft Visual Basic Technology logo are trademarks of Microsoft Corporation in the United States and other countries.

All other brand names, product names or trademarks belong to their respective holders.

## Third Party Software Program Credits

ACIS® Copyright © 1994, 1997, 1999 Spatial Technology, Inc., Three-Space Ltd., and Applied Geometry Corp. All rights reserved.

Copyright © 1997 Microsoft Corporation. All rights reserved.

International CorrectSpell™ Spelling Correction System © 1995 by Lernout & Hauspie Speech Products, N.V. All rights reserved.

InstallShield™ 3.0. Copyright © 1997 InstallShield Software Corporation. All rights reserved.

Portions Copyright © 1991-1996 Arthur D. Applegate. All rights reserved.

Portions of this software are based on the work of the Independent JPEG Group.

Typefaces from the Bitstream® typeface library copyright 1992.

Typefaces from Payne Loving Trust © 1996. All rights reserved.

The license management portion of this product is based on Élan License Manager © 1989, 1990, 1998 Élan Computer Group, Inc. All rights reserved.

Autodesk would like to acknowledge and thank Perceptual Multimedia, Inc., for the creative and technical design and the development of the Visual LISP Garden Path tutorial.

## GOVERNMENT USE

Use, duplication, or disclosure by the U. S. Government is subject to restrictions as set forth in FAR 12.212 (Commercial Computer Software-Restricted Rights) and DFAR 227.7202 (Rights in Technical Data and Computer Software), as applicable.

# Contents

	<b>Introduction . . . . .</b>	<b>1</b>
	<b>AutoLISP and Visual LISP . . . . .</b>	<b>2</b>
	What Visual LISP Offers . . . . .	2
	Working with Visual LISP and AutoCAD . . . . .	3
	<b>Using Visual LISP Documentation . . . . .</b>	<b>4</b>
	<b>Document Conventions . . . . .</b>	<b>7</b>
	<b>Related Documents . . . . .</b>	<b>8</b>
<b>Part I</b>	<b>Using the Visual LISP Environment . . . . .</b>	<b>9</b>
<b>Chapter 1</b>	<b>Getting Started. . . . .</b>	<b>11</b>
	Starting Visual LISP . . . . .	12
	Exploring the Visual LISP User Interface . . . . .	13
	Introducing the Visual LISP Text Editor . . . . .	14
	Other Visual LISP Windows. . . . .	15
	Touring the Visual LISP Menus. . . . .	15
	Variable Menu Contents . . . . .	15
	Visual LISP Menu Summary. . . . .	16
	Understanding the Console Window . . . . .	18
	Understanding the Visual LISP Text Editor . . . . .	19
	Loading and Running AutoLISP Programs . . . . .	20
	Running Selected Lines of Code . . . . .	21
	Using Extended AutoLISP Functions . . . . .	22
	Exiting Visual LISP . . . . .	22

<b>Chapter 2</b>	<b>Developing Programs with Visual LISP</b>	<b>23</b>
	Getting Organized	24
	Using the Console Window	25
	Understanding Console Behavior	25
	Using the Console Window with Multiple Drawings	28
	Using the Console Shortcut Menu	28
	Logging Console Window Activity	29
	Using the Text Editor	31
	Editing a File	31
	Using the Text Editor Shortcut Menu	32
	Using Keyboard Shortcuts in the Text Editor	33
	Moving and Copying Text	36
	Searching for Text	37
	Bookmarking Text	40
	Using Console and Editor Coding Aids	42
	Understanding Visual LISP Color Coding	42
	Using the Apropos Feature	43
	Letting Visual LISP Help You Complete Words	47
	Getting Help for AutoLISP Functions	50
	Formatting Code with Visual LISP	50
	Understanding Visual LISP Formatting Styles	51
	Applying Formatting Options	54
	Applying Visual LISP Comment Styles	58
	Saving and Restoring Formatting Options	58
	Formatter Restrictions	59
	Formatting Shortcut Keys	59
	Checking for Syntax Errors	61
	Checking the Balance of Parentheses	61
	Using Color Coding to Detect Syntax Errors	63
	Using the Check Command to Look for Syntax Errors	64
<b>Chapter 3</b>	<b>Debugging Programs</b>	<b>67</b>
	Introducing Visual LISP Debugging Features	68
	Learning by Example	69
	Stepping through the Debugging Example	69
	Using the Visual LISP Debugging Features	75
	Starting a Debugging Session	76
	Understanding Break Loops	76
	Using Breakpoints	78
	Using Visual LISP Data Inspection Tools	82
	Using the Watch Window	83
	Understanding the Trace Stack Window	85
	Using the Symbol Service Dialog Box	92
	Using Inspect Windows	95

	Viewing AutoCAD Drawing Entities . . . . .	104
<b>Chapter 4</b>	<b>Building Applications . . . . .</b>	<b>109</b>
	Compiling and Linking Programs. . . . .	110
	Using the Compiler . . . . .	110
	Compiling a Program from a File . . . . .	111
	Walking through a Compile Example. . . . .	113
	Loading and Running Compiled Programs. . . . .	114
	Linking Function Calls. . . . .	116
	Making Application Modules . . . . .	117
	Creating a New Application. . . . .	117
	Loading and Running Visual LISP Applications . . . . .	124
	Changing Application Options . . . . .	124
	Rebuilding an Application. . . . .	125
	Updating an Application . . . . .	125
	Designing for a Multiple Document Environment. . . . .	126
	Understanding Namespaces. . . . .	126
	Running an Application in Its Own Namespace. . . . .	129
	Sharing Data between Namespaces . . . . .	133
	Handling Errors in an MDI Environment . . . . .	134
	Limitations on Using AutoLISP in an MDI Environment. . . . .	136
<b>Chapter 5</b>	<b>Maintaining Visual LISP Applications. . . . .</b>	<b>137</b>
	Managing Multiple LISP Files . . . . .	138
	Understanding Visual LISP Projects. . . . .	138
	Defining a Project . . . . .	140
	Assigning Project Properties. . . . .	141
	Using the Project Window to Work with Project Files . . . . .	145
	Working with Existing Projects. . . . .	149
	Opening a Project . . . . .	149
	Finding a String in Project Source Files . . . . .	150
	Including a Project in a Visual LISP Application. . . . .	151
	Optimizing Application Code. . . . .	153
	Defining Build Options . . . . .	153
	Choosing a Compilation Mode . . . . .	155
	Choosing a Link Mode. . . . .	157
	Understanding Safe Optimization. . . . .	157
<b>Chapter 6</b>	<b>Working with ActiveX. . . . .</b>	<b>161</b>
	Using ActiveX Objects with AutoLISP. . . . .	162
	Understanding the AutoCAD Object Model. . . . .	162
	Object Properties . . . . .	164

Object Methods . . . . .	164
Collections of Objects . . . . .	164
Accessing AutoCAD Objects . . . . .	165
Using the Inspect Tool to View Object Properties . . . . .	166
Moving Forward from the Application Object . . . . .	168
Summarizing the Process . . . . .	168
Performance Considerations . . . . .	169
Using Visual LISP Functions with ActiveX Methods . . . . .	170
Determining the Visual LISP Function You Need . . . . .	171
Determining How to Call a Function . . . . .	172
Converting AutoLISP Data Types to ActiveX Data Types . . . . .	173
Viewing and Updating Object Properties . . . . .	180
Using ActiveX Methods That Return Values in Arguments . . . . .	184
Listing an Object's Properties and Methods . . . . .	185
Working with Collection Objects . . . . .	187
Releasing Objects and Freeing Memory . . . . .	189
Converting Object References . . . . .	190
Handling Errors Returned by ActiveX Methods . . . . .	191
Using ActiveX to Interact with Other Applications . . . . .	193
Importing a Type Library . . . . .	193
Establishing a Connection to an Application . . . . .	195
Coding a Sample Application . . . . .	196
Using ActiveX without Importing a Type Library . . . . .	198

## Chapter 7    **Advanced Topics . . . . . 201**

Attaching Reactors to AutoCAD Drawings . . . . .	202
Understanding Reactor Types and Events . . . . .	202
Defining Callback Functions . . . . .	205
Creating Reactors . . . . .	207
Working with Reactors in Multiple Namespaces . . . . .	210
Querying, Modifying, and Removing Reactors . . . . .	211
Transient versus Persistent Reactors . . . . .	215
Reactor Use Guidelines . . . . .	216

## Part II    **Using the AutoLISP Language . . . . . 219**

### Chapter 8    **AutoLISP Basics . . . . . 221**

AutoLISP Expressions . . . . .	222
AutoLISP Function Syntax . . . . .	223
AutoLISP Data Types . . . . .	224
Integers . . . . .	224
Reals . . . . .	225

Strings . . . . .	225
Lists . . . . .	225
Selection Sets . . . . .	226
Entity Names . . . . .	226
VLA-objects . . . . .	226
File Descriptors . . . . .	226
Symbols and Variables . . . . .	227
AutoLISP Program Files . . . . .	229
Formatting AutoLISP Code . . . . .	229
Comments . . . . .	230
Color Coding . . . . .	231
AutoLISP Variables . . . . .	231
Displaying the Value of a Variable . . . . .	231
Nil Variables . . . . .	232
Predefined Variables . . . . .	232
Number Handling . . . . .	233
String Handling . . . . .	234
Basic Output Functions . . . . .	235
Displaying Messages . . . . .	236
Control Characters in Strings . . . . .	237
Wild-Card Matching . . . . .	238
Equality and Conditional . . . . .	240
List Handling . . . . .	240
Point Lists . . . . .	242
Dotted Pairs . . . . .	244
Symbol and Function Handling . . . . .	245
Using defun to Define a Function . . . . .	245
C:XXX Functions . . . . .	247
Local Variables in Functions . . . . .	251
Functions with Arguments . . . . .	252
Error Handling . . . . .	254
Using the *error* Function . . . . .	255
Catching Errors and Continuing Program Execution . . . . .	257
 Chapter 9   Using AutoLISP to Communicate with AutoCAD . . . . .	 259
Query and Command Functions . . . . .	260
Command Submission . . . . .	260
System and Environment Variables . . . . .	263
Configuration Control . . . . .	264
Display Control . . . . .	264
Controlling Menus . . . . .	264
Control of Graphics and Text Windows . . . . .	267
Control of Low-Level Graphics . . . . .	267
Getting User Input . . . . .	268

The getxxx Functions . . . . .	268
Control of User-Input Function Conditions. . . . .	271
Geometric Utilities . . . . .	274
Object Snap . . . . .	274
Text Extents. . . . .	275
Conversions . . . . .	279
String Conversions . . . . .	279
Angular Conversion . . . . .	281
ASCII Code Conversion. . . . .	282
Unit Conversion . . . . .	283
Coordinate System Transformations . . . . .	286
File Handling . . . . .	289
File Search . . . . .	289
Accessing Help Files. . . . .	290
Device Access and Control . . . . .	291
Accessing User Input . . . . .	291
Calibrating Tablets . . . . .	291

<b>Chapter 10 Using AutoLISP to Manipulate AutoCAD Objects . . . . .</b>	<b>295</b>
Selection Set Handling . . . . .	296
Selection Set Filter Lists. . . . .	298
Passing Selection Sets between AutoLISP and ObjectARX Applications. . . . .	305
Object Handling . . . . .	305
Entity Name Functions . . . . .	305
Entity Data Functions . . . . .	311
Entity Data Functions and the Graphics Screen. . . . .	319
Old-Style Polylines and Lightweight Polylines. . . . .	320
Non-Graphic Object Handling . . . . .	321
Extended Data—xdata . . . . .	323
Organization of Extended Data. . . . .	324
Registration of an Application . . . . .	327
Retrieval of Extended Data . . . . .	327
Attachment of Extended Data to an Entity . . . . .	329
Management of Extended Data Memory Use. . . . .	329
Handles in Extended Data. . . . .	330
Xrecord Objects . . . . .	331
Symbol Table and Dictionary Access . . . . .	332
Symbol Tables . . . . .	332
Dictionary Entries . . . . .	334



<b>Part III</b>	<b>Working with Programmable Dialog Boxes. . . . .</b>	<b>335</b>
<b>Chapter 11</b>	<b>Designing Dialog Boxes . . . . .</b>	<b>337</b>
	Dialog Box Components . . . . .	338
	Using DCL to Define Dialog Boxes . . . . .	340
	The <i>base.dcl</i> and <i>acad.dcl</i> Files . . . . .	340
	Referencing DCL Files . . . . .	340
	DCL Syntax . . . . .	341
	Displaying Dialog Boxes with Visual LISP . . . . .	344
	Preview Error Handling . . . . .	346
	Semantic Auditing of DCL Files . . . . .	346
	Adjusting the Layout of Dialog Boxes . . . . .	347
	Distributing Tiles in a Cluster . . . . .	348
	Adjusting the Space between Tiles . . . . .	349
	Adjusting Space at the Right Side or Bottom . . . . .	350
	Fixing the Spacing Around a Boxed Row or Column . . . . .	350
	Customizing Exit Button Text . . . . .	351
	Design Guidelines . . . . .	353
	Aesthetics and Ergonomics . . . . .	353
	Consistent Design and Clear Language . . . . .	353
	User Control . . . . .	353
	Forgiving Errors . . . . .	355
	Providing Help . . . . .	355
	Users with Disabilities . . . . .	356
	Using Capitalization . . . . .	356
	Avoiding Abbreviations . . . . .	357
	Layout . . . . .	357
	Size and Placement . . . . .	357
	Disabling Tiles . . . . .	358
	Nesting Dialog Boxes . . . . .	358
	Hiding Dialog Boxes . . . . .	358
	Providing Defaults . . . . .	359
	Handling Keyboard Input . . . . .	359
	International Language Considerations . . . . .	359
	Guidelines for Predefined Tiles and Clusters . . . . .	360
	Error Handling . . . . .	363
<b>Chapter 12</b>	<b>Managing Dialog Boxes . . . . .</b>	<b>365</b>
	Controlling Dialog Boxes with AutoLISP Programs . . . . .	366
	Quick Overview . . . . .	366
	Functions Restricted When a Dialog Box Is Open . . . . .	367
	Action Expressions and Callbacks . . . . .	368
	Action Expressions . . . . .	369

Callback Reasons . . . . .	370
Handling Tiles. . . . .	372
Initializing Modes and Values. . . . .	372
Changing Modes and Values at Callback Time . . . . .	373
Handling Radio Clusters . . . . .	374
Handling Sliders . . . . .	375
Handling Edit Boxes . . . . .	376
Nesting Dialog Boxes . . . . .	376
Hiding Dialog Boxes. . . . .	377
Requesting a Password . . . . .	379
List Boxes and Pop-Up Lists . . . . .	380
List Operations . . . . .	380
Processing List Elements . . . . .	382
Image Tiles and Buttons . . . . .	383
Creating Images. . . . .	383
Handling Image Buttons . . . . .	385
Application-Specific Data . . . . .	386
DCL Error Handling . . . . .	386
Setting the Auditing Level to Affect Error Messages . . . . .	387
Dialog Box Function Summary . . . . .	388
Function Sequence . . . . .	388
The Sample Block Definition Dialog Box . . . . .	389

## Chapter 13 Programmable Dialog Box Reference . . . . . 391

Tile Attributes . . . . .	392
Attribute Types . . . . .	392
Restricted Attributes . . . . .	393
User-Defined Attributes. . . . .	393
Synopsis of Predefined Attributes. . . . .	394
DCL Attribute Catalog . . . . .	397
Functional Synopsis of DCL Tiles. . . . .	408
Predefined Active Tiles . . . . .	408
Tile Clusters. . . . .	408
Decorative and Informative Tiles . . . . .	409
Text Clusters . . . . .	409
Dialog Box Exit Buttons and Error Tiles . . . . .	410
Restricted Tiles. . . . .	410
DCL Tile Catalog. . . . .	410
Programmable Dialog Box Function Synopsis. . . . .	427
Dialog Box Opening and Closing Functions . . . . .	427
Tile- and Attribute-Handling Functions . . . . .	428
List Box and Pop-Up List-Handling Functions . . . . .	428
Image Tile-Handling Functions. . . . .	429
Application-Specific Data-Handling Functions . . . . .	429

<b>Part IV</b>	<b>Appendixes</b>	<b>431</b>
<b>Appendix A</b>	<b>AutoLISP Function Synopsis</b>	<b>433</b>
	Category Summary	434
	Basic Functions	436
	Application-Handling Functions	436
	Arithmetic Functions	437
	Equality and Conditional Functions	439
	Error-Handling Functions	440
	Function-Handling Functions	441
	List Manipulation Functions	442
	String-Handling Functions	444
	Symbol-Handling Functions	445
	Utility Functions	446
	Conversion Functions	446
	Device Access Functions	447
	Display Control Functions	448
	File-Handling Functions	449
	Geometric Functions	450
	Query and Command Functions	451
	User Input Functions	452
	Selection Set, Object, and Symbol Table Functions	453
	Extended Data-Handling Functions	453
	Object-Handling Functions	453
	Selection Set Manipulation Functions	455
	Symbol Table and Dictionary-Handling Functions	455
	Memory Management Functions	457
	Visual LISP Extensions to AutoLISP	457
	ActiveX Collection Manipulation Functions	458
	ActiveX Data Conversion Functions	458
	ActiveX Method Invocation Functions	459
	ActiveX Object-Handling Functions	460
	ActiveX Property-Handling Functions	460
	Curve Measurement Functions	461
	Dictionary Functions	462
	Object-Handling Functions	463
	Reactor Functions	464
	VLX Namespace Functions	467
	Namespace Communication Functions	468
	Windows Registry Functions	469

<b>Appendix B</b>	<b>Visual LISP Environment and Formatting Options. . . . .</b>	<b>471</b>
	Window Attributes . . . . .	472
	Syntax Coloring. . . . .	472
	Configure Current. . . . .	473
	Set Current Window to Prototype . . . . .	475
	Set All Windows to Prototype. . . . .	475
	Font . . . . .	475
	Environment Options. . . . .	475
	General Options . . . . .	475
	Visual LISP Format Options . . . . .	479
	Page Format Options. . . . .	483
	Save Settings . . . . .	484
 <b>Appendix C</b>	 <b>AutoLISP Error Codes . . . . .</b>	 <b>485</b>
	Error Codes . . . . .	486
	 Index . . . . .	 491

# Introduction

For years, AutoLISP® has set the standard for customizing AutoCAD®. Now Visual LISP™ (VLISP)™ represents the next generation of LISP for AutoCAD, by adding significantly more capabilities. VLISP extends the language to interface with objects via the Microsoft ActiveX™ Automation interface, and enhances AutoLISP's ability to respond to events through the implementation of reactor functions. As a development tool, VLISP provides a complete, integrated development environment (IDE) that includes a compiler, debugger, and other tools to increase productivity when customizing AutoCAD.

## In This Chapter

- AutoLISP and Visual LISP
- Using Visual LISP Documentation
- Document Conventions
- Related Documents

# AutoLISP and Visual LISP

AutoLISP is a programming language designed for extending and customizing AutoCAD functionality. It is based on the LISP programming language, whose origins date back to the late 1950s. LISP was originally designed for use in Artificial Intelligence (AI) applications, and is still the basis for many AI applications.

AutoCAD introduced AutoLISP as an application programming interface (API) in Release 2.1, in the mid-1980s. LISP was chosen as the initial AutoCAD API because it was uniquely suited for the unstructured design process of AutoCAD projects, which involved repeatedly trying different solutions to design problems.

Visual LISP (VLISP) is a software tool designed to expedite AutoLISP program development. The VLISP integrated development environment (IDE) provides features to help ease the tasks of source-code creation and modification, program testing, and debugging. In addition, VLISP provides a vehicle for delivering standalone applications written in AutoLISP.

In the past, developing AutoLISP programs for AutoCAD meant supplying your own text editor for writing code, then loading the code into AutoCAD and running it. Debugging your program meant adding statements to print the contents of variables at strategic points in your program. You had to figure out where in your program to do this, and what variables you needed to look at. If you discovered you still didn't have enough information to determine the error, you had to go back and change the code again by adding more debugging points. And finally, when you got the program to work correctly, you needed to either comment out or remove the debugging code you added.

## What Visual LISP Offers

During the development cycle of an AutoLISP application or routine, the AutoLISP user performs a number of operations that are not available within the AutoCAD software. Some of these operations—like text editing—are available with other software tools. Others, such as full AutoLISP source-level debugging, are introduced only with VLISP. In VLISP, you perform most of the necessary operations inside a single environment. This permits text editing, program debugging, and interaction with AutoCAD and other applications.

The following are components of the Visual LISP IDE:

- Syntax Checker recognizes erroneous AutoLISP constructs and improper arguments in calls to built-in functions.
- File Compiler improves the execution speed and provides a secure and efficient delivery platform.
- Source Debugger, designed specifically for AutoLISP, supports stepping through AutoLISP source code in one window while simultaneously displaying the results of code execution in an AutoCAD drawing window.
- Text file editor uses AutoLISP and DCL color coding, as well as other AutoLISP syntax support features.
- AutoLISP Formatter restructures programs into an easily readable format.
- Comprehensive Inspect and Watch features provide convenient access to variable and expression values for data structure browsing and modification. These features may be used to explore AutoLISP data and AutoCAD drawing entities.
- Context-sensitive Help provides information for AutoLISP functions and a powerful Apropos feature for symbol name search.
- Project Management system makes it easy to maintain multiple-file applications.
- Compiled AutoLISP files are packaged into a single module.
- Desktop Save and Restore capabilities preserve and reuse the windowing environment from any VLISP session.
- Intelligent Console window introduces a new level of convenience and efficiency for AutoLISP users. The basic functions of the Console correspond to the AutoCAD Text Screen functions and provide a number of interactive features, such as history scrolling and full-input line editing.

## Working with Visual LISP and AutoCAD

VLISP contains its own set of windows and menus that are distinct from the rest of AutoCAD, but VLISP does not run independently of AutoCAD. Whenever you work in VLISP, AutoCAD must also be running. When you run AutoLISP programs from the VLISP IDE, you will usually need to interact with the AutoCAD graphics or Command windows to respond to program prompts.

If AutoCAD is minimized when VLISP turns control over to it, you must manually restore and activate the AutoCAD window to continue. VLISP will not restore the AutoCAD window for you. Instead, a Visual LISP symbol appears in the VLISP window and remains there until you activate AutoCAD and respond to the prompts at the AutoCAD Command prompt. The “Getting Started” chapter shows an example of this; see “Loading and Running AutoLISP Programs” on page 20.



# Using Visual LISP Documentation

The *Visual LISP Developer's Guide* explains how to use the VLISP IDE and how to build and run AutoLISP applications. This guide also introduces the constructs of the AutoLISP language.

All users should check the AutoCAD Readme file for notices concerning AutoLISP and Visual LISP. The Readme file is in the AutoCAD Help directory and contains information that became available too late to be included in this manual.

If you have developed AutoLISP applications in earlier releases of AutoCAD, it is important that you refer to the Readme file for information on AutoLISP changes that may affect your applications.

Additional AutoLISP and Visual LISP manuals are available online through the Visual LISP and AutoCAD Help menus:

- The *AutoLISP Reference* briefly describes and provides examples of every AutoLISP function. Refer to the *AutoLISP Reference* when you need to look up the syntax of a function, or determine what a function returns.
- The *Visual LISP Tutorial* contains step-by-step instructions guiding you toward building a working Visual LISP application.

This *Visual LISP Developer's Guide* assumes you have some experience with AutoCAD and have basic user-level skills with Microsoft Windows®. Prior experience with AutoLISP is not required, but it will make your learning curve much smoother.

The *Visual LISP Developer's Guide* is divided into the following sections:

- “Using the Visual LISP Environment” describes how to use VLISP to develop and test AutoLISP programs.
- “Using the AutoLISP Language” is a detailed guide describing the elements and structures of the AutoLISP language.
- “Working with Programmable Dialog Boxes” describes how to design and implement dialog boxes in your AutoLISP applications.
- “Appendixes” includes a function synopsis summarizing AutoLISP functions by category, information on AutoLISP error codes, and a summary of the environment and program options available in Visual LISP.

If you are already familiar with AutoLISP, you may want to begin the *Visual LISP Tutorial* after reading chapter 1. This is a matter of personal comfort: if you feel you need to understand how everything works before using a tool,



you might be better off reading some or all of chapters 2 through 7 of this guide before trying the tutorial.

Following are some guidelines to help you get the most out of the *Visual LISP Developer's Guide*:

- Begin by reading chapter 1, “Getting Started.” This chapter tells you how to invoke VLISP from AutoCAD, identifies what you’ll see when VLISP first starts, and describes how to load and run existing AutoLISP programs from VLISP. Chapter 1 introduces and briefly describes the windows you will be working with in the VLISP IDE. Use this chapter to orient yourself to the VLISP environment.
- If you do not already know AutoLISP, read all of chapter 8, “AutoLISP Basics,” and at least browse chapters 9 and 10, “Using AutoLISP to Communicate with AutoCAD” and “Using AutoLISP to Manipulate AutoCAD Objects,” respectively. After that, you can either work through the tutorial or read more chapters in the *Visual LISP Developer's Guide*.
- To search for a function that meets a particular programming need, refer to appendix A, “AutoLISP Function Synopsis,” in this guide.

The following table summarizes the organization of this manual:

Chapter organization		
Chapter	Title	Contents
1	Getting Started	Orientation to Visual LISP
2	Developing Programs with Visual LISP	Shows you how to use the VLISP text editor to enter AutoLISP program code, format the code, and check the code for AutoLISP syntax errors. Also shows you how to run the code you’ve developed from the VLISP editor window.
3	Debugging Programs	Shows you how to use VLISP to trace program execution, watch the value of variables change during program execution, see the sequence in which expressions are evaluated, and step through program execution one instruction at a time.
4	Building Applications	Introduces the VLISP file compiler and shows how you can use the VLISP Application Wizard to build standalone applications.

## Chapter organization (continued)

Chapter	Title	Contents
5	Maintaining Visual LISP Applications	Describes how to define VLISP projects and use them to simplify working with multi-file applications. This chapter also explains compiler optimization features, and how to use them in a project.
6	Working with ActiveX	Describes how to use ActiveX objects with VLISP, and how to access other applications through ActiveX.
7	Advanced Topics	Describes how to attach reactors to AutoCAD drawings and objects.
8	AutoLISP Basics	Introduces basic AutoLISP concepts, such as how to use expressions and variables, handle numbers and strings, display output, build lists, and define functions.
9	Using AutoLISP to Communicate with AutoCAD	Describes AutoLISP functions that you can use to issue AutoCAD commands and to interact with users in the AutoCAD environment.
10	Using AutoLISP to Manipulate AutoCAD Objects	Describes AutoLISP functions you can use to manipulate AutoCAD drawing entities, selections sets, extended data, and symbol tables.
11	Designing Dialog Boxes	Introduces the elements that make up dialog boxes. Explains DCL file structure and syntax, and presents AutoLISP and DCL code that defines and displays a sample dialog box.
12	Managing Dialog Boxes	Describes how to control dialog boxes with AutoLISP programs.
13	Programmable Dialog Box Reference	Lists and describes all the DCL tiles and their associated attributes, and summarizes the AutoLISP functions available for work with programmable dialog boxes.

# Document Conventions

This document follows a number of stylistic and naming conventions when describing actions or defining terms. Often, distinct typefaces are used to distinguish items from the rest of the text. The following table lists some of the conventions used in the *Visual LISP Developer's Guide*.

Typographical conventions	
Text element	Example
Program code examples are displayed in 8 pt Courier font.	(defun initfunc ( ) (setq *GLOBAL-WARNING* 4.0))
Text you enter is shown in boldface.	At the VLISP Console prompt, enter shape
File names and directory names are shown in <i>italics</i> when referred to in a sentence.	Double-click the file name <i>drawline.lsp</i> . The default install directory is <i>C:\Program Files\AutoCAD\VLISP</i>
In definitions that include variable text, the variable is in <i>italics</i> .	A FAS file named <i>appname-init.fas</i> , where <i>appname</i> is the application module name
AutoLISP and Visual LISP variable names are in courier type.	Double-click on any occurrence of the variable name origin-y The *LAST-VALUE* system variable
AutoLISP function names are shown in bold courier type.	Use the <b>entget</b> function... Use <b>vlax-import-type-library</b> to import

# Related Documents

In addition to the *AutoLISP Reference* and the *Visual LISP Tutorial*, several other AutoCAD publications may be required by users building applications with Visual LISP:

- *AutoCAD ActiveX and VBA Reference* contains information on accessing ActiveX methods, properties, and objects. If you develop AutoLISP applications that use ActiveX automation to reference AutoCAD objects, you will need to refer to this reference guide. It is available through the AutoCAD and Visual LISP Help menus.
- *AutoCAD Customization Guide* contains basic information on creating customized AutoCAD applications. For example, it includes information on creating customized menus, linetypes, and hatch patterns. The *Customization Guide* is available through the AutoCAD and Visual LISP Help menus.
- The *DXF Reference* describes drawing interchange format (DXF™) and the DXF group codes that identify attributes of AutoCAD objects. You may need to refer to the *DXF Reference* when working with association lists describing entity data. The *DXF Reference* is available through the AutoCAD and Visual LISP Help menus.
- The *ObjectARX Reference* contains information on using ObjectARX™ to develop customized AutoCAD applications. AutoCAD reactor functionality is implemented through ObjectARX. If you develop AutoLISP applications that implement reactor functions, you may want to refer to this manual.

The *ObjectARX Reference* is not included when you install AutoCAD. To obtain the manual, download the ObjectARX SDK (Software Development Kit) from the Autodesk World Wide Web site (<http://www.autodesk.com>). Click on the Product Index link and look for ObjectARX under Development Tools.

# **Part 1**

## **Using the Visual LISP Environment**





# Getting Started

This chapter introduces you to the look and feel of the Visual LISP interactive development environment (IDE), and shows you how to run AutoLISP programs in Visual LISP.

## In This Chapter

# 1

- Starting Visual LISP
- Exploring the Visual LISP User Interface
- Touring the Visual LISP Menus
- Understanding the Console Window
- Understanding the Visual LISP Text Editor
- Loading and Running AutoLISP Programs
- Exiting Visual LISP

# Starting Visual LISP

The Visual LISP (VLISP) interactive development environment runs in a separate set of windows from the rest of AutoCAD. You must explicitly start VLISP to work in the interactive development environment.

To start Visual LISP

- 1 Start AutoCAD.
- 2 Choose Tools ► AutoLISP ► Visual LISP Editor from the AutoCAD menu, or enter the following at the Command prompt:

`vlisp`

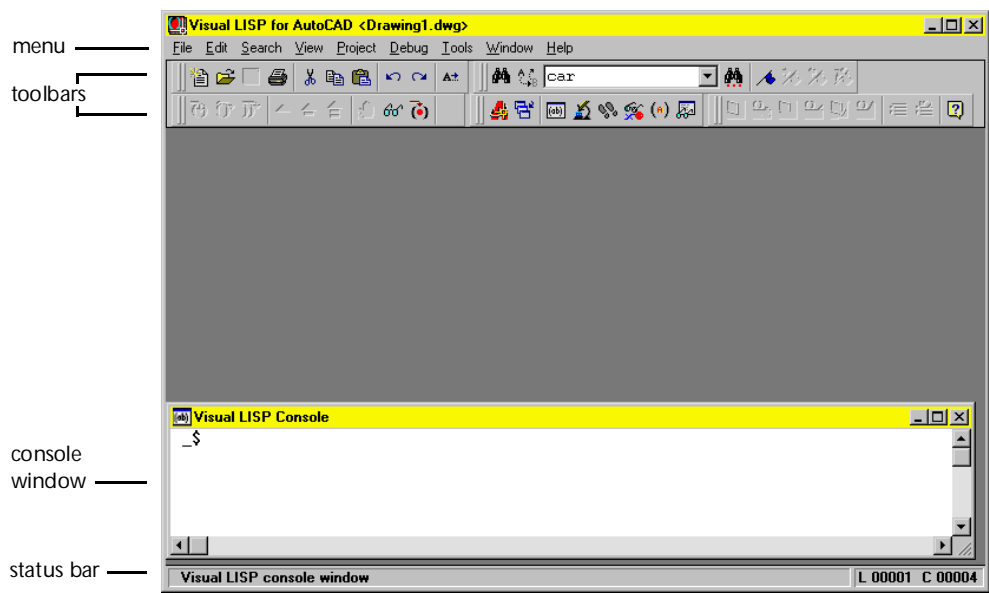
You can use either the menu or the `vlisp` command to return to the VLISP IDE at any time.

Note that AutoCAD also recognizes the `vlide` command to start or return to Visual LISP. This command name stands for “Visual LISP interactive development environment.” AutoCAD issues the `vlide` command to call VLISP, and as a result you may sometimes see “VLIDE” displayed in the AutoCAD Command window.




# Exploring the Visual LISP User Interface

When you first start VLISP, it displays the following screen:



The areas shown in the VLISP screen are as follows:

- |          |  |
|----------|--|
| Menu     | You can issue VLISP commands by choosing from the various menu items. If you highlight an item on a menu, VLISP displays a brief description of the command's function in the status bar at the bottom of the screen.  |
| Toolbars | Click toolbar buttons to issue VLISP commands quickly. There are five toolbars—Debug, Edit, Find, Inspect, and Run—each representing a distinct functional group of VLISP commands. (In the figure shown on this page, the toolbars are adjacent to one another, each toolbar beginning with an icon (  ). You can execute many, but not all, menu commands from the toolbars. If you move your mouse pointer over a toolbar button and leave it there for a couple of seconds, VLISP displays a tooltip indicating the function of the button. A more descriptive explanation appears in the status bar at the bottom of the VLISP screen. |

Console Window	This is a separate, scrollable window within the main VLISP window. In the Console window, you can type AutoLISP commands, similar to the way you do in the AutoCAD Command window. You can also issue many Visual LISP commands from this window, instead of using the menu or toolbars. See “Understanding the Console Window” on page 18 for more information about the Console window.
Status Bar	The information displayed in the status bar located at the bottom of the screen varies according to what you are doing in VLISP.

You may also see a minimized Trace window. During startup, this window contains informational messages about the current release of VLISP, and may contain additional information if VLISP encounters errors during startup.

## Introducing the Visual LISP Text Editor

You will spend much of your time in VLISP creating or modifying AutoLISP programs. VLISP comes with an integrated text editor for you to use with AutoLISP code.

To see how the text editor window displays code, open a sample AutoLISP program. Begin with the *drawline.lsp* file provided with VLISP.

---

**NOTE** The sample files are only included in your installation if you chose Full install, or if you chose Custom install and selected the Samples item. If you previously installed AutoCAD and did not install the samples, rerun the install, choose Custom, and select only the Sample item.

---

To view a LISP program in the VLISP text editor

- 1 From the VLISP menu, choose File ► Open File.
- 2 In the Open File dialog box, select the *Sample \ VisualLISP* folder, which is in your AutoCAD install directory.
- 3 Double-click the *drawline.lsp* file.

VLISP opens the file in a new window—the text editor window—and displays the name of the file in the status bar. If you make a change to the text in the editor window, or add new text, VLISP places an asterisk (\*) next to the file name in the status bar. The asterisk remains next to that file name until you either save your changes or close the file.

You can work on more than one file at a time. Each time you open a file, VLISP displays the file in a new text editor window.

## Other Visual LISP Windows

VLISP displays some output in the Console window, but several VLISP functions create their own windows in which to display results. For example, when you trace the sequence of events during a running AutoLISP program, the Trace function opens a window and displays program events. You cannot enter text in these output windows, but you can copy text from them and paste the text into the editor or Console windows.

## Touring the Visual LISP Menus

You can issue VLISP commands by choosing from the various menu items. For example, from the File menu you can create a new AutoLISP program file, select an existing program file to edit, and print the file you're editing.

### Variable Menu Contents

Menu contents may vary depending on which VLISP window (for example, text editor, Console) is active. To activate a different window, click in the window's title bar, or in any empty area of that window.

As an example, click in the text editor window containing the *drawline.lsp* file, then choose Edit from the VLISP menu. You'll see the following list:

Undo	Ctrl-Z
Redo	Ctrl-Alt-Z
Cut	Ctrl-X
Copy	Ctrl-C
Paste	Ctrl-V
Delete	Del
Select All	Ctrl-A
Parentheses Matching	Ctrl-M ▶
Extra commands	Ctrl-E ▶

Note that the last items on the menu are Parentheses Matching and Extra Commands.

Now click in the title bar of the VLISP Console window, then select the Edit menu item again:

Undo	Ctrl-Z
Redo	Ctrl-Alt-Z
Cut	Ctrl-X
Copy	Ctrl-C
Paste	Ctrl-V
Delete	Del
Select All	Ctrl-A
Clear Console Window	
Clear Console Input	Esc
Keep but Ignore Input	Shift-Esc
Parentheses Matching	Ctrl-M ▶
Console History Up	Tab
Console History Down	Shift-Tab

Notice that Extra Commands is no longer the last item on the menu. Parentheses Matching is followed by two new items, Console History Up and Console History Down; these items apply only to a Console window.

## Visual LISP Menu Summary

The following table summarizes the VLISP menu items.

### Visual LISP menu items

Menu item	Uses
File	Create a new AutoLISP program file for editing, open an existing file, save changes to program files, build Visual LISP application files, and print program files.
Edit	Copy and paste text, undo the last change you made to text (or undo the last command entered in the Console window), select text in the VLISP editor or Console windows, match parentheses in expressions, and redisplay previous commands entered in the Console window. See the chapter titled “Developing Programs with Visual LISP” for more information on Edit features.
Search	Find and replace text strings, set bookmarks, and navigate among bookmarked text. See “Using the Text Editor” on page 31 for information on these topics.
View	Find and display the value of variables and symbols in your AutoLISP code. For more information on this topic, see chapter 3, “Debugging Programs.”

## Visual LISP menu items (*continued*)

Menu item	Uses
Project	Work with projects and compile programs. See “Managing Multiple LISP Files” on page 138, and “Compiling and Linking Programs” on page 110 for information on these topics.
Debug	Set and remove breakpoints in your program and step through program execution one expression at a time. You can then check the state of variables and the results of expressions. See chapter 3, “Debugging Programs,” for more information on these features.
Tools	Set VLISP options for text formatting and various environment options, such as the placement of windows and toolbars.
Window	Organize the windows currently displayed in your VLISP session, or activate another VLISP or AutoCAD window.
Help	Display online help.

# Understanding the Console Window

From the VLISP Console window, you can enter and run AutoLISP commands and see the results. This is similar to what you can do in the AutoCAD Command window, but there are a few differences—some subtle—in how you accomplish the same task in these two windows. For example, to display the current value of an AutoLISP variable in VLISP, you simply type the variable name in the Console window and press ENTER. To view the value of a variable in AutoCAD, you must precede the variable name with an exclamation point (!) when you type it in the Command window.

The Console window is also where VLISP displays AutoLISP diagnostic messages and the results of many AutoLISP functions. For example, output from the `print` and `princ` functions is displayed in the Console window. You can scroll through the Console window to view previously entered text and output.

For a description of VLISP Console window features, see “Using the Console Window” on page 25. The following is a brief summary of these features:

- Evaluating AutoLISP expressions and displaying the return values from those expressions.
- Entering AutoLISP expressions on multiple lines by pressing CTRL + ENTER to continue on the next line.
- Evaluating multiple expressions at one time.
- Copying and transferring text between the Console and text editor windows. Most text editor commands are also available in the Console window.
- Retrieving previous commands entered in the Console by pressing TAB. You can press TAB repeatedly to retrieve earlier commands. Press SHIFT + TAB to reverse the direction of command retrieval.
- Performing an associative search through the input history by pressing TAB. For example, if you begin an expression with (+ and then press TAB, VLISP retrieves the last command you entered that begins with (+. To reverse the direction of the search, press SHIFT + TAB.
- Pressing ESC clears any text following the Console prompt.
- Pressing SHIFT + ESC leaves the text you entered at the Console prompt without evaluating the text, and displays a new Console prompt.
- Right-clicking or pressing SHIFT + F10 anywhere in the Console window displays a menu of VLISP commands and options. For example, you can use this feature to copy and paste text in the Console command line, search for text, and initiate VLISP debugging features.

Note that if you type text at the Console prompt but switch to the AutoCAD window before pressing ENTER, the text will no longer be at the prompt when you return to the VLISP window.

## Understanding the Visual LISP Text Editor

The VLISP text editor is much more than a writing tool, it's a central component of the VLISP programming environment. To appreciate the versatility and value of the VLISP text editor, you need to be familiar with the AutoLISP language. If you are not yet familiar with AutoLISP, you can learn the basics in chapter 8, "AutoLISP Basics," and find additional information in chapter 9, "Using AutoLISP to Communicate with AutoCAD," and chapter 10, "Using AutoLISP to Manipulate AutoCAD Objects."

Here are some of the major features of the text editor:

Color Coding of Files	The text editor identifies different parts of an AutoLISP program and assigns distinct colors to them. This allows you to find program components easily such as function calls and variable names, and helps you find typographical errors.
Formatting of Text	The text editor can format AutoLISP code for you, making the code easier to read. You can choose from a number of different formatting styles.
Parenthesis Matching	AutoLISP code contains many parentheses, and the editor helps you detect missing parentheses by finding the close parenthesis that goes with an open parenthesis.
Execution of AutoLISP Expressions	You can test expressions and lines of code without leaving the text editor.
Multiple File Searching	The text editor can search for a word or expression in several files with a single command.
Syntax Checking of AutoLISP Code	The text editor can evaluate AutoLISP code and highlight syntax errors.

Details on using the VLISP text editor begin with "Using the Text Editor" on page 31.

# Loading and Running AutoLISP Programs

Once you have opened an AutoLISP program file in the VLISP text editor, you can load and run it. Loading is the process by which functions in a program file are made available to the VLISP command interpreter. You can try this out with the *drawline.lsp* sample program.

To load and run a program in a Visual LISP text editor window

- 1 Make sure the text editor window containing the *drawline.lsp* program is active. If you are not sure whether the window is active, click anywhere in the window to activate it.



- 2 Choose the Load Active Edit Window button from the Run toolbar, or choose Tools ► Load Text in Editor from the VLISP menu.

VLISP responds by displaying a message in the Console window indicating it has loaded the program.

- 3 Run the `drawline` function from the Console prompt by entering the function name in parentheses, then pressing ENTER:

```
_$ (drawline)
```

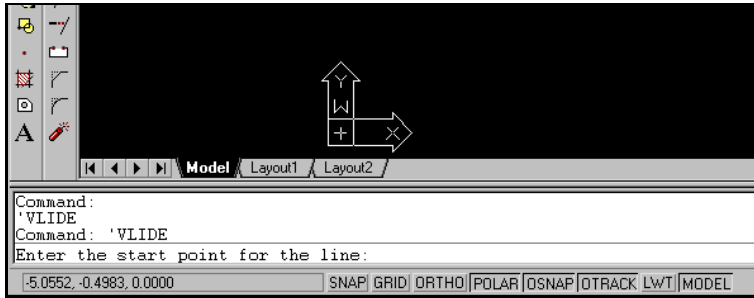
The `drawline` function will ask you to specify two points, and will then draw a straight line between those points. When `drawline` asks for user input, VLISP turns control over to AutoCAD to prompt you for the points. What you see next depends on whether or not the AutoCAD windows are currently displayed on your desktop. If AutoCAD is already on your desktop, you'll see the AutoCAD windows. But if AutoCAD is currently minimized on your desktop, the windows won't automatically be restored and displayed. Instead, VLISP remains visible and your mouse pointer changes to a VLISP symbol:



This symbol indicates that the VLISP window is no longer active. If this is the case, you must manually switch to the AutoCAD window. Click the AutoCAD icon on the Windows task bar to activate AutoCAD.

The following figure shows AutoCAD prompting for the input requested by `drawline`:





- 4 Respond to the prompts by specifying points in the graphics window or on the Command line.

After you respond to the prompts, control returns to VLISP and you will once again see the VLISP window.

When you enter commands in the VLISP Console window or run a program loaded from the text editor, you may be frequently switching back and forth between the VLISP and AutoCAD windows. Aside from using the standard Windows methods of switching between windows, you can activate the AutoCAD window by choosing Window ► Activate AutoCAD from the VLISP menu, or by pressing the Activate AutoCAD button on the Run toolbar. If you are in AutoCAD and want to return to the VLISP environment, you can enter `vlisp` at the Command prompt, or choose:

Tools ► AutoLISP ► Visual LISP Editor from the AutoCAD menu.



## Running Selected Lines of Code

With VLISP, you can select lines of code in the text editor window and run only the selected code, instead of the whole program.

To run selected lines of AutoLISP code in a Visual LISP editor window

- 1 Using the *drawline.lsp* program as an example, highlight the following lines of code:

```
(setq pt1(getpoint "\nEnter the start point for the line: "))
      pt2(getpoint pt1 "\nEnter the end point for the line: "))
```



- 2 Press the Load Selection button on the Run toolbar.

VLISP immediately runs the code and switches control to AutoCAD to prompt you for input.

## Using Extended AutoLISP Functions

VLISP provides some extensions to the AutoLISP language that are not loaded automatically when you start AutoCAD. These functions have names that begin with *vla-*, *vla-*, and *vlr-*. The *vla-* functions implement ActiveX support in AutoLISP (described in chapter 6, “Working with ActiveX”). The *vla-* functions provide ActiveX utility and data conversion functions, dictionary handling functions, and curve measurement functions. The *vlr-* functions provide support for AutoCAD reactors. Before you can use any of these functions, you need to load the AutoLISP extensions with the following function call:

```
(vl-load-com)
```

This function first checks whether or not the AutoLISP extensions are already loaded; if so, the function does nothing, otherwise it loads the extensions.

If you’re using the VLISP interactive development environment to develop code, chances are you’ll want to use the AutoLISP extensions at some point. It’s a good practice to issue `vl-load-com` when you start VLISP, or to include a call to the function in your *acad.dcl* file, so that it loads automatically. But if you write programs that use any of the extended AutoLISP functions, you need to call `vl-load-com` in those programs to ensure that the functions are available to other users running your code.

## Exiting Visual LISP

When you are finished with your VLISP session, you can close the program by either choosing Exit from the File menu or clicking the Windows Close button. Note that AutoCAD does not completely unload VLISP but merely closes all VLISP windows.

Upon exiting AutoCAD, if you have made any changes to the code in any VLISP text editor window and have not saved those changes, you will be asked if you want to save your changes. You can either save all the changes you’ve made by choosing Yes, or save none of the changes by choosing No.

VLISP retains its state when you exit. The next time you start a VLISP session, VLISP automatically opens whichever files and windows were open when you last exited.

# Developing Programs with Visual LISP

VLISP provides many tools and features that help you develop AutoLISP programs. This chapter describes the features that help you during the coding phase of program development. These features format AutoLISP code to improve readability and help you detect syntax errors in your code.

## In This Chapter

# 2

- Getting Organized
- Using the Console Window
- Using the Text Editor
- Using Console and Editor Coding Aids
- Formatting Code with Visual LISP
- Checking for Syntax Errors

# Getting Organized

To develop an AutoLISP program with VLISP you must perform the following steps:

- Think about which tasks you want to accomplish with your program, and how to approach those tasks.
- Design the program.
- Write the code.
- Format the code for readability.
- Check for errors in the program.
- Test and debug the program.

This chapter provides you with information to help you accomplish writing, formatting, and checking tasks. The “Debugging Programs” chapter describes the debugging features of VLISP. The “Building Applications” and “Maintaining Visual LISP Applications” chapters describe how to package your programs into applications that can be run by other users, and how to organize application components to facilitate future updates.

# Using the Console Window

Most programming in VLISP takes place within the confines of the VLISP text editor, but the ability to program interactively with AutoLISP provides some unique advantages to the development process. In the VLISP Console window you can enter AutoLISP code and immediately see the results of executing that code.

Enter text in the Console window following the Console prompt, which looks like the following:

```
_$
```

VLISP saves the text you enter and any output from executing the text. You can then scroll through the Console window and see what transpired. You can copy any text in the window and paste it at the Console prompt or in another Windows application.

## Understanding Console Behavior

The VLISP Console window is similar in some respects to the AutoCAD Command window, but it provides many more features. Although the Console window and the AutoCAD Command window provide similar capabilities, you don't always use the same process to accomplish identical tasks. For example, to display the current value of an AutoLISP variable in VLISP, you simply type the variable name in the Console window and press ENTER. To view the value of a variable in AutoCAD, you must precede the variable name with an exclamation point (!) when you enter it at the AutoCAD Command prompt.

Unlike the AutoCAD Command window, where pressing SPACEBAR causes expression evaluation, text input at the VLISP Console prompt is not processed until you press ENTER. This permits you to do the following in the Console window:

- Continue an AutoLISP expression on a new line. To continue entering an expression on a new line, press CTRL + ENTER at the point you want to continue.
- Input more than one expression before pressing ENTER. VLISP evaluates each expression before returning a value to the Console window.
- If you select text in the Console window (for example, the result of a previous command or a previously entered expression), then press ENTER. VLISP copies the selected text at the Console prompt.

The VLISP Console window and the AutoCAD Command window differ in the way they process the SPACEBAR and TAB keys. In the VLISP Console window, a space plays no special role and serves only as a separator. In the AutoCAD Command window, pressing the SPACEBAR outside an expression causes AutoCAD to process the text immediately, as if you had pressed ENTER.

## Using the Console Window History

You can retrieve text you previously entered in the Console window by pressing TAB while at the Console prompt. Each time you press TAB, the previously entered text replaces the text at the Console prompt. You can repeatedly press TAB until you cycle through all the text entered at the Console prompt during your VLISP session. After you've scrolled to the first entered line, VLISP starts again by retrieving the last command entered in the Console window, and the cycle repeats. Press SHIFT+TAB to scroll the input history in the opposite direction.

For example, assume you entered the following commands at the Console prompt:

```
(setq origin (getpoint "\nOrigin of inyn sign: "))
(setq radius (getdist "\nRadius of inyn sign: " origin))
(setq half-r (/ radius 2))
(setq origin-x (car origin))
(command "_ . CIRCLE" origin radius)
```

### To retrieve commands entered in the Console window

- 1 Press TAB once. VLISP retrieves the last command entered and places it at the Console prompt:

```
_$ (command "_ . CIRCLE" origin radius)
```

- 2 Press TAB again. The following command displays at the Console prompt:

```
_$ (setq origin-x (car origin))
```

- 3 Press TAB again. VLISP displays the following command:

```
_$ (setq half-r (/ radius 2))
```

- 4 Now press SHIFT+TAB. VLISP reverses direction and retrieves the command you entered after the previous command:

```
_$ (setq origin-x (car origin))
```

- 5 Press SHIFT+TAB again. VLISP displays the following command:

```
_$ (command "_ . CIRCLE" origin radius)
```

This was the last command you entered at the Console prompt.

- 6 Press SHIFT+TAB again. Because the previous command retrieved was the last command you entered during this VLISP session, VLISP starts again by retrieving the first command you entered in the Console window:

```
_$ (setq origin (getpoint "\nOrigin of input sign: "))
```

Note that if you enter the same expression more than once, it appears only once as you cycle through the Console window input history.

You can perform an associative search in the input history to retrieve a specific command that you previously entered.

### To perform an associative search of the Console input history

- 1 Enter the text you want to locate. For example, enter (command at the Console prompt:

```
_$ (command
```

- 2 Press TAB. VLISP searches for the last text you entered that began with (command:

```
_$ (command "_ . CIRCLE" origin radius)
```

If VLISP does not find a match, it does nothing (except possibly emit a beep). Press SHIFT+TAB to reverse the direction of the associative search and find progressively less-recent inputs.

## Interrupting Commands and Clearing the Console Input Area

To interrupt a command entered in the Console window, press SHIFT+ESC. For example, if you enter an invalid function call like the following:

```
_$ ((setq origin-x (car origin)
((_)>
```

Pressing SHIFT+ESC interrupts the command, and VLISP displays an “input discarded” message like the following:

```
((_)> ; <input discarded>
_$
```

(Note that in this example, you can also complete the command by entering the missing close parentheses.)

If you type text at the Console prompt, but do not press ENTER, then pressing ESC clears the text you typed. If you press SHIFT+ESC, VLISP leaves the text you entered in the Console window but displays a new prompt without evaluating the text.

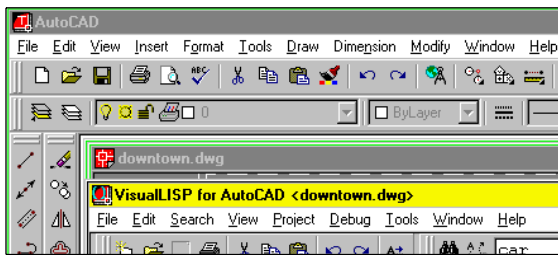
If you type part of a command at the Console prompt, but activate the AutoCAD window before pressing ENTER, VLISP displays a new prompt when you next activate the VLISP window. The text you typed is visible in the Console window history, so you can copy and paste it, but you cannot retrieve

the text by pressing TAB, because it was not added to the Console history buffer.

## Using the Console Window with Multiple Drawings

There is a single Console window for all open AutoCAD drawing documents. When you scroll through the Console window, you see commands entered in all document contexts. This differs from the AutoCAD Command window, which shows only the commands issued against the current drawing. In other words, each AutoCAD drawing has its own Command window, but all drawings share the same VLISP Console prompt.

VLISP automatically switches context when you change the active drawing document in AutoCAD. The active document in AutoCAD is always the active document in VLISP. Commands entered in the VLISP Console window always apply to the active document. The title bar of the VLISP window contains the name of the active drawing document. The following example shows how the title bars of the AutoCAD and VLISP windows identify the same drawing document *downtown.dwg*:



## Using the Console Shortcut Menu

The most important functions needed when working with the VLISP Console window are combined into a shortcut menu for fast access. Right-click anywhere in the Console window or press SHIFT + F10 to display the shortcut menu.

Depending on whether there is text selected in the Console window and depending on the cursor position, some commands may not be appropriate at the moment and cannot be activated from the shortcut menu. The follow-



ing table summarizes the commands that may be available from the Console window shortcut menu.

Console window shortcut menu commands	
Command	Action
Cut	Removes the selected text from the Console window and moves it to the Windows Clipboard
Copy	Copies the selected text to the Clipboard
Paste	Pastes the Clipboard contents to the cursor location
Clear Console window	Empties the Console window
Find	Finds specified text in the Console window
Inspect	Opens the Inspect dialog box
Add Watch	Opens the Watch window
Apropos window	Opens the Apropos window
Symbol Service	Opens the Symbol Service dialog box
Undo	Reverses the last operation
Redo	Reverses the effects of the previous Undo
AutoCAD Mode	Transfers all input to the AutoCAD command line for evaluation
Toggle Console Log	Copies Console window output to the log file

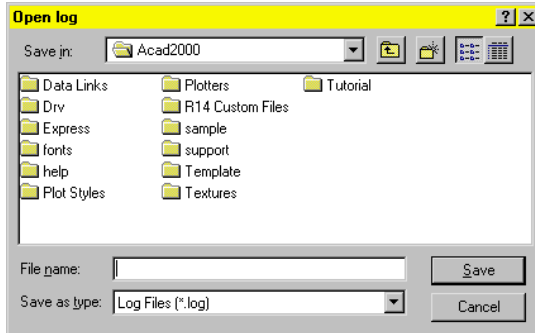
Note also that you can cut and paste text between the VLISP Console window and the AutoCAD Command window.

## Logging Console Window Activity

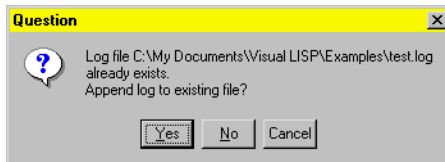
You can keep a record of all Console window activity by logging the activity in a file. Later, you can view the file and review the activity that occurred in the Console window.

## To control Console logging activity

- 1 Create a log file by choosing **File ► Toggle Console Log** from the VLISP menu. Note that the Console window must be active for the Toggle Console Log option to be available.



- 2 Choose a directory for the log file and specify a file name for the log. If the file already exists, VLISP prompts you with the following message:



If you reply Yes, VLISP appends future Console window information to the current contents of the file. If you reply No, VLISP overwrites the file and its original contents will be lost. Choose Cancel to terminate the operation and specify a different file name.

- 3 To close the log file and quit the logging process, choose **Toggle Console Log** from the File menu again.

The state of Console window logging is indicated in the Console window's title bar. If logging is in effect, VLISP displays the name of the log file in the title bar. If logging is off, no file name appears in the title bar.

If you do not close the log file before exiting VLISP, it closes the log file automatically upon exit. After a log file is closed, you can view its contents with any text editor, such as the VLISP text editor.

# Using the Text Editor

If you just need to run a few simple AutoLISP expressions, entering the expressions in the VLISP System Console may suffice. For anything more than that, however, you will need to use the VLISP text editor and save your AutoLISP code in a file.

The text editor is a basic component of VLISP. It is easy to use and, if you have some experience using Windows, you can begin using it after a quick review of this chapter.

The VLISP text editor has a number of features designed to support AutoLISP programming, such as selecting of complete AutoLISP expressions, matching of balanced parentheses, syntax coloring, and executing AutoLISP expressions without leaving the editor window. Most text editor commands can be called from the menu bar, and some of the most frequently used commands are also available from toolbar push buttons.

## Editing a File



To open a new file in the VLISP text editor, choose **File ► New File** from the menu bar. An empty editor window appears on the screen, and you can begin entering text.

To start a new line, press **ENTER**. The text editor does not wrap your text when it reaches the end of the visible text editor window, so everything you type goes on the same line until you press **ENTER**.

You can indent lines of text manually, but VLISP automatically formats code for you as you enter new lines of code. You can also copy text from another file and have VLISP format the block of text you add. See “Formatting Code with Visual LISP” on page 50 for details on using the VLISP code formatting features.

## Undoing Your Last Change



You can reverse your last edit action by choosing **Edit ► Undo** from the VLISP menu bar. You can undo a virtually unlimited number of changes, back to the point at which you last saved the file.



To reverse the effects of the Undo command, choose **Edit ► Redo** from the menu. Redo only works if you issue it immediately after Undo.

## Creating Automatic Backup Files

VLISP supports the automatic creation of backup copies of files loaded by the text editor. The actual backup creation occurs when you save the file for the

first time. Backup files have the same name as your original file, except the file extension begins with an underscore (\_) and is followed by the first two characters of the original extension. For example, the backup file for *drawline.lsp* would be *drawline.\_ls*.

Automatic creation of backup files is an option you can set by choosing **Tools ► Environment Options ► General Options**. Choose the Editor option labeled “Backup the File Edited on First Save” to turn on automatic backup. By default, this option is already selected when you first install VLISP.

## Restoring from a Backup File

If a backup file exists, you can restore the file you are editing to its original content, reversing all the changes you made. From the File menu, choose **Revert** to restore the file. If there is no backup file for the text in the editor window, VLISP displays an error message.

## Editing an Existing File



Choose **File ► Open** from the VLISP menu to open an existing file. VLISP opens a new text editor window for the file you select. You can open any number of files and work on them simultaneously. VLISP places each file in its own editor window.

---

**NOTE** If you select text in any VLISP window before choosing **File ► Open**, the selected text is placed in the File name field of the Open dialog box.

---

When you exit VLISP, it notes which files are open and saves this information for your next VLISP session. The next time you start VLISP, it automatically opens the files for you.

## Using the Text Editor Shortcut Menu

Right-clicking your mouse in an active VLISP text editor window brings up a shortcut menu for quick access to frequently used commands. Depending on whether there is text highlighted in the editor window and depending on the position of the cursor, some commands on the shortcut menu may be inactive. The following table summarizes the editor shortcut commands:

### Text editor window shortcut menu commands

Command	Action
Cut	Moves the selected text to the Clipboard
Copy	Copies the selected text to the Clipboard
Paste	Pastes the Clipboard contents to the cursor position
Find	Finds the specified text in one or more editor windows
Go to Last Edited	Moves the cursor to the position you last edited
Toggle Breakpoint	Sets a breakpoint at the cursor position, or removes a breakpoint if one is set currently at that position
Inspect	Opens the Inspect dialog box
Add Watch	Opens the Watch window
Apropos window	Opens the Apropos window
Symbol Service	Opens the Symbol Service dialog box
Undo	Reverses the last operation
Redo	Reverses the effects of the previous Undo

## Using Keyboard Shortcuts in the Text Editor

The VLISP text editor provides numerous keyboard shortcuts you can use instead of menu selections and mouse clicks.

### Correcting Text

You can delete words or lines using the following shortcuts:

### Text correction shortcut keys

To	Press
Erase a word to the left of the cursor	CTRL + BACKSPACE
Erase a word to the right of the cursor	SHIFT + BACKSPACE
Delete characters from the cursor position to the end of the current line	CTRL + E, then press E

You can also use the overstrike mode to insert text. Overstrike mode is toggled on and off by pressing **INSERT**. When in overstrike mode, each character you type replaces existing text. The cursor changes shape from vertical to horizontal when in overstrike mode.

## Selecting Text

The simplest method to select text is to double-click your left mouse button. The amount of text selected depends on the location of your cursor.

- If the cursor immediately precedes an open parenthesis, VLISP selects all the following text up to the matching close parenthesis.
- If the cursor immediately follows a close parenthesis, VLISP selects all preceding text up to the matching open parenthesis.
- If the cursor immediately precedes or follows a word, or is within a word, VLISP selects that word.

To select specific text, press and hold the **SHIFT** key while pressing the direction (arrow) keys on the keyboard. Other keyboard methods of text selection are listed in the following table:

Text selection shortcut keys	
To	Press
Expand the selection to the next line	SHIFT + DOWN ARROW
Expand the selection to the previous line	SHIFT + UP ARROW
Expand the selection to the end of the line	SHIFT + END
Expand the selection to the beginning of the line	SHIFT + HOME
Expand the selection down one window, or abandon selection of the next window, if it is currently selected	SHIFT + PAGEDOWN
Expand the selection up one window, or abandon selection of the previous window, if it is currently selected	SHIFT + PAGEUP
Expand the selection to the next word, or abandon selection of the next word, if it is currently selected	CTRL + SHIFT + RIGHT ARROW
Expand the selection to the previous word, or abandon selection of the previous word, if it is currently selected	CTRL + SHIFT + LEFT ARROW

### Text selection shortcut keys (*continued*)

To	Press
Expand the selection up to the matching left parenthesis	CTRL + SHIFT + [
Expand the selection up to the matching right parenthesis	CTRL + SHIFT + ]
Move the cursor to the other side of the selection	ALT + ENTER

## Navigation Shortcuts

In addition to using the cursor arrow keys, you can use the following VLISP editor shortcuts to navigate through your text:

### Navigation shortcut keys

To move	Press
One word to the left	CTRL + LEFT ARROW
One word to the right	CTRL + RIGHT ARROW
To the end of a line	END
To the beginning of a line	HOME
Down one window	PAGEDOWN
Up one window	PAGEUP
To the end of a document	CTRL + END
To the start of a document	CTRL + HOME
To the matching left parenthesis	CTRL + [
To the matching right parenthesis	CTRL + ]

## Indenting Shortcuts

Most indenting of program code is best handled by the VLISP automatic code formatting and Smart Indent features, and by customizing the formatter's options (see "Formatting Code with Visual LISP"). But there are some things you may want to do by yourself.

To indent selected lines of code, press **TAB** or press **CTRL + E**, and then choose **Indent Block**. VLISP inserts a **TAB** character at the beginning of each line you selected. You can control the indent amount of the **TAB** character by choosing **Tools ► Window Attributes ► Configure Current** and setting the **Tab Width** value. You can also use the following keyboard shortcuts to adjust the indentation of text.

#### Code indentation shortcuts

To	Do
Adjust the indent of the current selection to the preceding AutoLISP text.	Press <b>SHIFT + TAB</b>
Clear trailing <b>SPACE</b> and <b>TAB</b> characters, insert a new line, and indent at the level of the previous non-empty line.	Press <b>SHIFT + ENTER</b>
Insert a new line without clearing trailing <b>SPACE</b> and <b>TAB</b> characters of the current line.	Press <b>CTRL + ENTER</b>

## Moving and Copying Text

In addition to using the standard Windows **Cut**, **Copy**, and **Paste** functions, the VLISP text editor allows you to drag text from one location to another within the edit window.

### To move text by dragging

- 1 Select the text you want to move.
- 2 Point anywhere inside the selected area, and press and hold the left mouse button.
- 3 Drag the text to the new location.
- 4 Release the mouse button.

To copy the text instead of moving it, follow the same steps, but press **CTRL** before releasing the mouse button in step 4.

You can also take selected text and copy it into a new file. With the text selected, press **CTRL + E** to display a list of options, and choose **Save Block As**. VLISP replies by displaying a dialog box for you to specify where you want to save the text.

VLISP uses the Windows Clipboard for all cut and copy operations. Therefore, you can exchange text with any other Windows application that



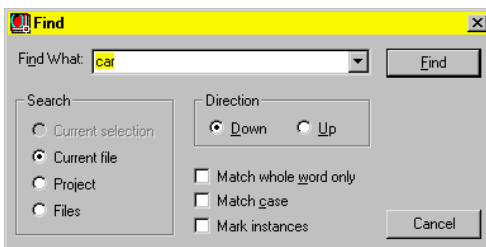
supports these functions. This also means you can copy and paste text between the text editor and the VLISP Console window.

Remember that immediately after moving or copying text, you can change your mind and reverse the action, using the Undo function.

## Searching for Text



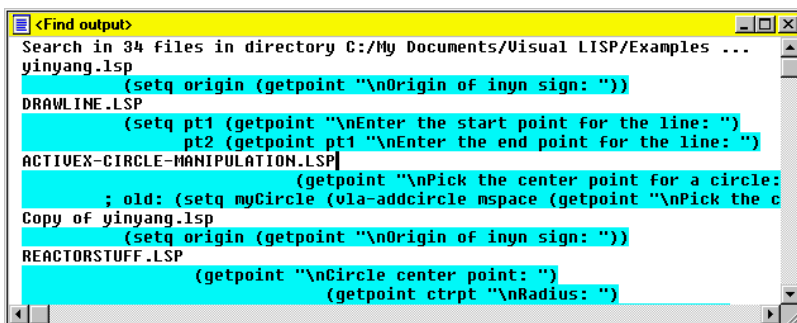
The VLISP text editor has extensive text-searching capabilities. From the Search menu, choose Find to begin a search, or press the Find toolbar button. VLISP displays the Find dialog box:



In the Find What data entry field, type the character string you want to locate. If there is text selected when you enter the Find command, this text is automatically placed in the Find What field.

Choose Find to start the search. When searching through a single file, press F3 to search for the next occurrence of your search string. Choose Cancel to end the search.

When searching through multiple files (see the next topic, “Choosing Search Options”), VLISP displays the matches it found in an output window like the following:



Double-click on any highlighted lines in the Find Output window to open the associated LISP file in a VLISP editor window.

## Choosing Search Options

Under the Search heading, indicate the extent of the search you want VLISP to conduct. You can choose one of the following:

Current Selection	Searches only the text highlighted in the editor window.
Current File	Searches through the entire file in the active editor window.
Find in Project	With this option selected, VLISP prompts you to specify the name of the VLISP project you want to search. It will search all the files in this project and display all matches in a new output window. See “Finding a String in Project Source Files” on page 150 for more information on this option.
Find in Files	If you select this option, VLISP allows you to specify a Windows directory (folder) to search for the text. Optionally, you can instruct VLISP to search all subdirectories of that directory as well. VLISP will search through all the files and display all matches in a new output window.

When searching for text within the current file, the Direction setting determines where VLISP looks next for the search text. Choose Down to search forward (toward the end of the file) from the cursor position. Choose Up to search backward (toward the beginning of the file) from the cursor position.

The Find dialog box also includes the following options:

Match Whole Word Only	If selected, VLISP will only match complete words. For example, if the search term is <i>ent</i> and VLISP encounters the word <i>enter</i> in the text, VLISP does not consider this a match. However, if the Match Whole Word Only option is not selected, VLISP considers the <i>ent</i> within <i>enter</i> to be a match.
Match Case	If selected, VLISP only matches text set in the same case. In this instance, <i>Ent</i> and <i>ent</i> are not considered a match. If Match Case is not selected, <i>Ent</i> and <i>ent</i> are considered a match.

**Mark Instances** If you select this option, the position of the located text will be added to the bookmark ring (see “Bookmarking Text” on page 40). This lets you return quickly to this code position later. Searches that find all occurrences of a string add each position to the bookmark ring.

## Repeating an Earlier Search

VLISP saves each search string you enter in a pull-down list on the toolbar.

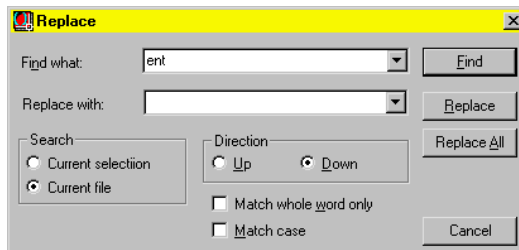


To repeat a search you made earlier, click the pull-down arrow and select a search term from the toolbar list. Press the Find Toolbar String button to conduct the search.

## Replacing Text



The Search menu contains a Replace function that is used to replace the search text with a text string that you specify.



The Replace dialog box is similar to the Find dialog box, but with fewer options. It contains an additional Replace With entry field, in which you specify the text you want VLISP to substitute for the search text. Specify the search text in the Find What field.

You can take the following actions from the Replace dialog box:

- Press Find Next to find the next occurrence of the search string.
- Press Replace to replace the found text with the replacement string.  
If you don't want to replace this occurrence of the text, press Find Next to search for the next occurrence of the text, or Cancel to end the search.
- Press Replace All to replace all occurrences of the search string with the replacement string.
- Press Cancel to end the Replace function.

## Bookmarking Text

The bookmark feature helps you navigate through VLISP text editor windows by letting you mark up to 32 positions (bookmarks) in each window. Once 32 bookmarks are set, adding a new bookmark results in the oldest bookmark being removed.

Each text editor window maintains its own set of bookmarks, and the bookmark navigation tools let you walk through the marks within each window independently of the other windows. A set of bookmarks within a window is known as a bookmark ring. You can step either forward or backward through the ring, and eventually return to the starting point.

Whenever you step to a bookmark, VLISP automatically places a marker at the location you are stepping from. In effect, the marker for the place you are jumping to is moved to the place you jumped from. This makes it easy to return to your original location just by stepping back in the opposite direction, or by cycling through all the bookmarks until you get back to the starting point.

To add a bookmark



- 1 Move the cursor to the location you want to mark.
- 2 Press the Toggle Bookmark button on the toolbar, or press ALT+. (ALT plus a period).

Bookmarks may also be inserted automatically when using the Find command to search for text. See the discussion on search options in “Searching for Text” on page 37 for more information on this feature.

### To move the cursor from one bookmark to the next



- 1 Move the cursor to the previous bookmark in the ring by choosing Search ► Bookmarks ► Previous Bookmark, or by pressing the Previous Bookmark toolbar icon. You can also accomplish this by pressing CTRL+, (CTRL plus a comma).



- 2 Move the cursor to the next bookmark in the ring by choosing Search ► Bookmarks ► Next Bookmark, or by pressing the Next Bookmark toolbar icon. You can also accomplish this by pressing CTRL+. (CTRL plus a period).

In addition to jumping between bookmarks, you can also jump and select the text between two bookmarks.

### To move the cursor and select text between bookmarks

- 1 Press CTRL+SHIFT+, (comma) to select the text between the current location and the next bookmark.
- 2 Press CTRL+SHIFT+., (period) to select the text between the current location and the previous bookmark.

Removing a bookmark is similar to setting a bookmark.

### To remove a bookmark

- 1 Move the cursor to the bookmarked location.
- 2 Press the Toggle Bookmark button, or press ALT+. (ALT plus a period).

The Toggle Bookmark command works as an on/off switch. If you issue the command when a bookmark is set, Toggle Bookmark turns it off. Issue the same command when there is no bookmark set, and Toggle Bookmark inserts a bookmark.



- 3 To remove all the bookmarks in the active window, press the Clear All Bookmarks button on the toolbar, or choose Search ► Bookmarks ► Clear All Bookmarks from the VLISP menu.

# Using Console and Editor Coding Aids

Several VLISP coding aids are useful at both the Console and text editor windows. One highly visual aid is the assignment of colors to AutoLISP language elements. Color coding helps you detect syntax errors in your code. VLISP also contains several features to help you determine the names of variables and functions that you need to refer to in your program, and shortcuts to online help for AutoLISP functions.

## Understanding Visual LISP Color Coding

As soon as you enter text in the VLISP Console or text editor windows, VLISP attempts to determine if the entered word is a built-in AutoLISP function, a number, a string, or some other language element. VLISP assigns every type of element its own color. This helps you detect missing quotes or misspelled function names. The default color scheme is shown in the following table.

**Default color coding scheme for AutoLISP code**

AutoLISP language element	Color
Built-in functions and protected symbols	Blue
Strings	Magenta
Integers	Green
Real numbers	Teal
Comments	Magenta, on gray background
Parentheses	Red
Unrecognized items (for example, user variables)	Black

You can change the default colors by choosing **Tools ► Window Attributes ► Configure Current** from the VLISP menu. See “Configure Current” on page 473 for more information on setting colors.

The VLISP text editor provides color coding for LISP files, DCL files, SQL files, and C++ language source files (see “LISP, FAS, and Other File Types” on page 138 for a list of file types recognized by VLISP). VLISP uses the file name extension to determine a file’s type, and then selects the color coding accordingly. You can change the color coding style associated with a file type by

choosing Tools ► Window Attributes ► Syntax Coloring from the VLISP menu. All text entered in the Console window is treated as AutoLISP code.

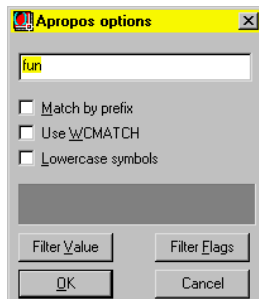
## Using the Apropos Feature

The Apropos feature is a tool that searches the VLISP symbol table. The symbol table contains every symbol read by the AutoLISP reader. This includes symbols in user programs and symbols that implement the AutoLISP language.

You can define specific search criteria for Apropos to use in searching the symbol table. For example, you can tell Apropos to search for all symbol names that contain a specific character string, and you can further refine that search to return only symbols that identify functions.



To invoke Apropos, choose View ► Apropos Window from the VLISP menu, or press the Apropos button on the VLISP toolbar. If you select text prior to invoking Apropos, VLISP immediately performs an Apropos search on the selected text. If no text is selected, VLISP displays the Apropos options dialog box:



In the input field of the Apropos options dialog box, enter the text you want Apropos to search for. The dialog box contains the following options:

Match by  
Prefix

If this option is turned on, Apropos searches for a match starting only from the first character of the symbol name. If the option is turned off, Apropos tries to match the text you entered starting at any position of a symbol name.

For example, with Match by Prefix off, a search on the word *get* returns symbol names including *get int*, *getpoint*, *ssget*, and *vla-getActive*. With Match by Prefix on, the same search does not return *ssget* and *vla-getActive*, because *get* appears in the middle of those symbol names, not at the beginning.

Use WCMATCH (wild card match)	If this option is turned on, Apropos treats asterisks as wild-card characters when searching. For example, if you specify <code>fun*</code> as the symbol you want matched, Apropos looks for all names that contain <i>fun</i> , no matter what characters follow. In contrast, with Use WCMATCH turned off, the asterisk is treated as a string and Apropos only matches names that precisely contain <code>fun*</code> .
Downcase Symbols	If turned on, any symbols you copy to the Clipboard with the Apropos service are converted to lowercase characters. If you paste the symbol name in another window, it appears in lowercase.
Filter Flags	This option lets you choose symbols with matching flag settings. VLISP displays a list of check boxes that correspond to the symbol flags described in “Understanding Symbol Flags” on page 94. If the flag filter is on, only symbols set with the selected flags are considered.
Filter Value	<p>Opens the Filter Value dialog box, from which you can select additional search criteria. You can choose one of the following:</p> <p><b>All</b> No filter.</p> <p><b>Null value</b> Only nil-valued symbols are considered for matching.</p> <p><b>Nonull value</b> Only symbols that are not nil are considered for matching.</p> <p><b>Functions</b> All function types (user-defined, built-in, etc.) are considered for matching.</p> <p><b>User function</b> Only user-defined functions (USUBR) are considered.</p> <p><b>Built-in function</b> Only built-in or compiled AutoLISP functions (SUBR) are considered for matching.</p> <p><b>Exrxsubr</b> Only external function names are matched.</p> <p>If you specify a filter value or filter flag, the message area of the Apropos options dialog box indicates your selections.</p> <p>When you’ve specified the criteria you want Apropos to use in its search, press OK to conduct the search.</p>

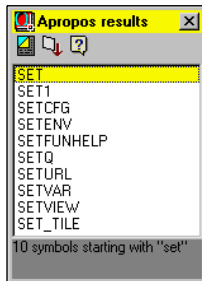


To search for AutoLISP symbols that begin with set

- 1 Choose View ► Apropos Window from the VLISP menu.
- 2 Enter set in the text input field of the Apropos options dialog box.
- 3 Select the Match by Prefix option.
- 4 Clear all other options in the Apropos options dialog box.
- 5 Press OK to conduct the search.

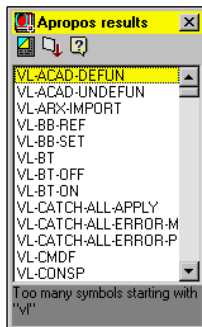
## Using the Results of an Apropos Search

Apropos displays the symbols matching your search criteria in the following window:



The bottom of the Apropos results window contains a message area with information about the results of the search. In the current example, the message indicates the number of symbols Apropos found in its search.

If the Apropos results window is not large enough to show all the symbols found, the window is displayed as scrollable. If the search returns over a thousand matches, Apropos will not be able to list all the symbols, even in a scrollable window. The message area in the results window warns you when this occurs, as in the following example from a search on the prefix VL:



If your search results in too many symbols for Apropos to display in the results window, you can use the Copy to Trace/Log feature to view the complete list of symbols in the VLISP Trace window.



To return to the Apropos options window and refine your search, press the Apropos options button in the Apropos results window toolbar. The toolbar also contains the following buttons:



**Copy to Trace/log** Copies the results of the Apropos search to the VLISP Trace window. Data in the Trace window can be copied using the Windows copy command. If Trace logging is active, the contents are also copied to the log file (see “Using Visual LISP Data Inspection Tools” on page 82 for information on Trace logging).



**Help** Invokes online help for the selected symbol. The symbol name is used as the Help index search value.

You can also use the Apropos results window’s shortcut menu on selected symbols. For example, you can select a symbol from the list and insert it into the VLISP Console or text editor window.

#### To insert a symbol from the Apropos results window

- 1 Select a symbol from the list.
- 2 Right-click to display the shortcut menu, and choose Copy to Clipboard from the list of options.
- 3 Click in the VLISP window at the point you want to insert the symbol name.
- 4 Right-click and select Paste from the context menu, or press CTRL + V to paste the text.

The other options on the shortcut menu are:

Inspect	Invoke the VLISP Inspect feature for the selected symbol. See “Using Inspect Windows” on page 95 for information on using this feature.
Print	Print the symbol name in the Console window. If you select a symbol name displayed in the Console window and press ENTER, VLISP copies the symbol name to the Console prompt.
Symbol	Invoke the VLISP Symbol Service feature for the selected symbol. See “Using the Symbol Service Dialog Box” on page 92 for information on using this feature.

Copy	Copy the selected symbol name to the *OBJ* system variable.
Add to Watch	Add the selected symbol to the Watch window. See “Using the Watch Window” on page 83 for information on using this feature.
Help	Invoke online help for the selected symbol.

## Letting Visual LISP Help You Complete Words

Two VLISP features, Complete Word by Match and Complete Word by Apropos, allow you to type part of a word and get help in completing the rest.

### Completing a Word by Matching

Using Complete Word by Match, VLISP completes a partially entered word by matching the part you have typed with another word in the same window. For example, suppose the following shows the history of your VLISP Console window:

```

_$ (setq origin (getpoint "\nOrigin of inyn sign: "))
_$ (setq radius (getdist "\nRadius of inyn sign: " origin))
_$ (setq half-r (/ radius 2))
_$ (setq origin-x (car origin))
_$ (command "_CIRCLE" origin radius)

```

In other words, these are the last five commands that you entered from the Console.

#### To complete a word by matching

- 1 Type the following at the Console prompt:

```
_$ (c
```

- 2 Press CTRL+SPACEBAR to invoke Complete Word by Match. VLISP finds the last word you entered that began with the letter “c,” and completes the word you started to type:

```
_$ (command
```

- 3 If that is not the word you are looking for, press CTRL+SPACEBAR again. VLISP searches back through the Console history for the previous occurrence of a word beginning with the letter “c”:

```
_$ (car
```

VLISP will keep searching for matching words each time you press CTRL+SPACEBAR. If you keep pressing CTRL+SPACEBAR after VLISP finds the last matching word, VLISP repeats the retrieval sequence. (Note that you can

also choose Search ► Complete Word by Match from the VLISP menu instead of pressing CTRL+SPACEBAR to invoke the Match feature.)

If VLISP does not find any matching words, it does nothing.

You can use Complete Word by Match in either the Console window or the text editor window. When you invoke the feature from the Console window, VLISP only searches the Console for a match; when invoked from a text editor window, VLISP only searches that editor window for a match.

The Complete Word by Match feature is not case-sensitive. In the previous example, you would have achieved the same result had you entered a capital C instead of a lowercase c.

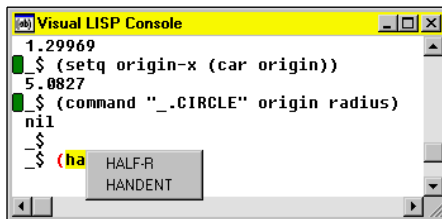
## Completing a Word by Apropos

With the Complete Word by Apropos feature, VLISP completes a partially entered word with a matching symbol name from the VLISP symbol table. To demonstrate this feature, assume you have entered the following commands shown in your VLISP Console window:

```
_$ (setq origin (getpoint "\nOrigin of inyn sign: "))
_$ (setq radius (getdist "\nRadius of inyn sign: " origin))
_$ (setq half-r (/ radius 2))
_$ (setq origin-x (car origin))
_$ (command "_ .CIRCLE" origin radius)
```

To use the Complete Word by Apropos feature

- 1 At the Console prompt, type the following:  
\_\$ (ha
- 2 Press CTRL+SHIFT+SPACE to invoke Complete Word by Apropos on the partially entered word. VLISP lists all symbol table entries that begin with “ha”:

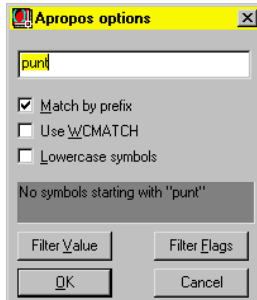


VLISP found two matching words in the symbol table. The half-r symbol is a variable you defined in the Console window, and the handent symbol represents an AutoLISP function.

- 3 Select the symbol you want to complete your typing. If you do not want to select a symbol, press ESC.

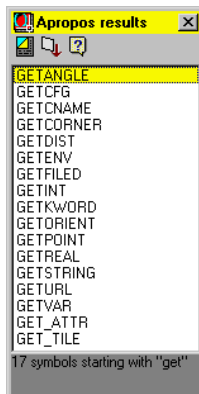
Note that you can also choose Search ► Complete Word by Apropos from the VLISP menu instead of pressing CTRL + SHIFT + SPACE to invoke the feature.

If no symbols match the text you’ve entered, VLISP displays the Apropos options dialog box:



The message area of the Apropos options dialog box shows the value that Apropos could not match. See “Using the Apropos Feature” on page 43 for information on setting Apropos options and renewing your search.

If VLISP finds more than 15 matching names in the symbol table, it displays the Apropos results dialog box. For example, type get at the Console prompt, then press CTRL + SHIFT + SPACEBAR to invoke the Apropos feature. VLISP displays the following dialog box:



You can select a symbol from the results window and copy it into your code using a shortcut menu. If you need additional help with copying the symbol to your program code, or using other features of the Apropos results window, see “Using the Results of an Apropos Search” on page 45.

## Getting Help for AutoLISP Functions



If you select a function name anywhere in a text editor or Console window, and then press the Help button on the Tools toolbar, VLISP displays help for the function. This feature works for any function recognized by VLISP. You can also press CTRL+F1 to view Help for the selected function.

## Formatting Code with Visual LISP

The VLISP code formatter arranges the text of AutoLISP expressions in a style that improves text appearance and readability. The code formatter includes a Smart Indent feature to arrange the indentation of program code automatically. The code formatter works automatically as you enter code in a VLISP text editor window. You can also explicitly invoke the formatter to rearrange selected blocks of text or all text in a VLISP editor window. This is useful for formatting text you copy from other editors, or for reformatting your code in a different style.

To format text in an active editor window



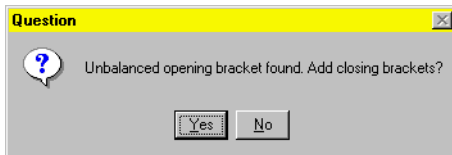
- 1 To format all the code in the window, choose Tools ► Format code in Editor from the VLISP menu, or click the Format Edit window button on the Tools toolbar.



- 2 To format only part of the code in the editor window, select a fragment of code text and choose Format Code In Selection from the Tools menu, or click the Format Selection button on the Tools toolbar.

If you select text to be formatted, the selection must contain valid AutoLISP expressions or the formatter will issue an error message.

If the formatter finds unbalanced parentheses in your code, it issues the following message:



Choose Yes to have VLISP add parentheses where it thinks they belong; choose No if you want to fix the parentheses on your own.

---

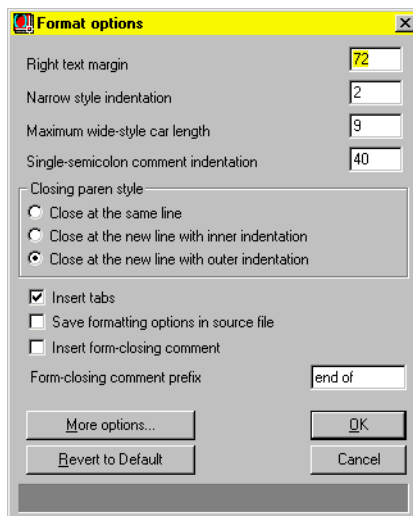
**NOTE** The VLISP formatter can balance the number of parentheses but usually does not insert the additional parentheses in the right places. See “Checking the Balance of Parentheses” on page 61 for more information on detecting and correcting unmatched parentheses.

---

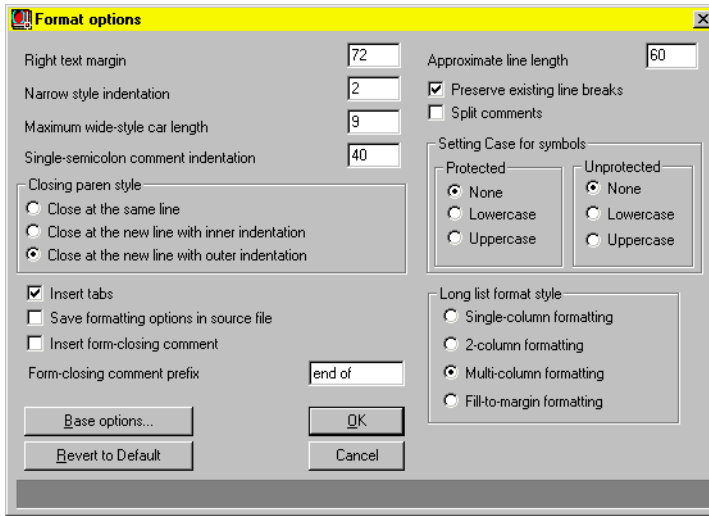
The VLISP Smart Indent feature works in the background as you type in the text editor. The indent is evaluated up to the current AutoLISP parenthesis nesting level. If the current expression is preceded by only a sequence of completed top-level AutoLISP expressions, the indentation will be zero. You can affect the amount of indentation by specifying Visual LISP format options; see the next two topics.

## Understanding Visual LISP Formatting Styles

The VLISP formatter chooses the appropriate formatting style according to rules that are explained in this section. You can influence the choice of VLISP styles through the options you set in the Format Options dialog box. To display the Format Options dialog box, choose **Tools ► Environment Options ► Visual LISP Format Options** from the VLISP menu:



Initially, VLISP displays only a subset of the formatting options you can specify. Press the More Options button in the Format Options dialog box to expand the window with additional formatting options.



The following are two main formatting style sets:

- A single-line formatting style—Plane
- Multiple-line formatting styles—Wide, Narrow, Column

The sample text below demonstrates the different formatting styles.

Sample text initial appearance:

```
(auto! oad " appl oad"
' (" appl oad"))
```

For a general function call expression, the formatter applies one of the styles in the following sections:

## Plane Style

In the Plane style, all arguments are placed in the same line, separated by a single space:

```
(auto! oad " appl oad" ' (" appl oad"))
```

The Plane style is applied to an expression when all the following conditions are met:

- The expression's last character position does not exceed the value of the Right Text Margin environment option.
- The expression's printing length is less than the value of the Approximate Line Length environment option (that is, last character position minus starting indentation position is less than this value).



- The expression does not contain embedded comments with Newline characters.

## Wide Style

In the Wide style, the first argument is placed in the same line as the function name, and other arguments are aligned in a column below the first argument.

```
(autoload "apload"
      ' ("apload")
)
```

The Wide style applies to an expression when the following conditions are met:

- The Plane style cannot be applied.
- The first element is a symbol, and the first element's length is less than the Maximum Wide Style Car Length environment option.

## Narrow Style

In the Narrow style, the first argument is placed on the next line after the function name, and other arguments are aligned in a column below the first argument. The displacement of the first argument's starting position relative to the expression starting position is controlled by the value of the Narrow Style Indentation environment option (in the following example, this value is equal to 2):

```
(autoload
  "apload"
  ' ("apload")
)
```

The Narrow formatting style applies for progn expressions, and for those instances when the Plane and Wide formatting styles cannot be applied.

## Column Style

In the Column style, all elements are positioned in a column. This style is appropriate for displaying quoted lists and COND-expression clauses. For example, the following text:

```
'((10 "{insertion}")      (1 "{string}")
  (7 "{style}"))
```

would be displayed as:

```
'((10 "{insertion}")
  (1 "{string}")
  (7 "{style}")
)
```

# Applying Formatting Options

In addition to affecting the basic formatting styles, you can choose from a number of Visual LISP format options.

## Close Parenthesis Style

This style controls the position of the close parenthesis for multiple-line formatting styles. You can select one of the following options:

Close at the Same Line	Close parenthesis on the last line of each formatting expression.
Close at the New Line with Inner Indentation	Close parenthesis on the next line following the last line of each formatting expression with the inner indent.
Close at the New Line with Outer Indentation	Close parenthesis on the next line following the last line of each formatting expression with the outer indent.

### Examples:

The initial expression is written as:

```
(cond ((/= (logand mask flg) 0)
      (list (list txton)))
)
```

Formatting result when Close at the Same Line option is selected:

```
(cond ((/= (logand mask flg) 0)
      (list (list txton))))
```

Formatting result when Close at the New Line with Inner Indentation is selected:

```
(cond ((/= (logand mask flg) 0)
      (list (list txton))
    )
)
```

Formatting result when Close at the New Line with Outer Indentation is selected:

```
(cond ((/= (logand mask flg) 0)
      (list (list txton))
    )
)
```

## Insert Form-Closing Comment

If you select this option, VLISP adds a comment following the close of an expression. However, the option takes effect only if the Close Parenthesis Style format setting is either Close at the New Line with Inner Indentation or Close at the New Line with Outer Indentation.

When the Insert Form-Closing Comment option is on, the VLISP formatter inserts a comment of the form

```
;_ end of <function name>
```

after each multiple-line function. This comment does not appear if an inline-comment, single-semicolon comment, or pasted-comment exists after the function call. You can change the comment text by entering a different comment in the Form-Closing Comment prefix field of the Format Options dialog box.

### Example

Initial text:

```
(autoarxload "image"  
  ("gi fi n" "pcxi n" "ri aspect" "ri backg" "ri edge"  
   "ri gamut" "ri grey" "ri thresh" "ti ffi n"))
```

Formatted text:

```
(autoarxload "image"  
  ("gi fi n"      "pcxi n"      "ri aspect"  
   "ri backg"     "ri edge"     "ri gamut"  
   "ri grey"      "ri thresh"   "ti ffi n"  
  )  
) ;_ end of autoarxload
```

Note the `_ end of autoarxload` comment in the last line of code.

## Preserve Existing Line Breaks

When the Preserve Existing Line Breaks option is on, the VLISP formatter inserts new lines whenever a new line is detected in the text it is formatting. When the option is off, the formatter can squeeze a multiple-line expression to the Plane style, if it fits within the right margin.

The following example shows how the Preserve Existing Line Breaks option works.

**Initial text:**

```
(if (/= s "Function canceled") (prin c (strcat
    "\nError: "
    s)) ; single semi colon cmt
)
```

**Formatting result if the option is on (default):**

```
(if (/= s "Function canceled")
    (prin c (strcat
        "\nError: "
        s
    )
    ) ; single semi colon cmt
)
```

**Formatting result when the option is off:**

```
(if (/= s "Function canceled")
    (prin c (strcat "\nError: " s)) ; single semi colon cmt
)
```

Note that multiple-line `prin c` and `strcat` expressions are compressed to a single line.

## Split Comments

When the Split Comments option is on, the formatter splits long comments that extend past the right margin.

For the previous example, if the Right Text Margin setting is 60, and Single-Semicolon comment indentation is 40, the formatter will split the comment as follows:

```
(if (/= s "Function canceled")
    (prin c (strcat "\nError: " s)) ; single
    ; semi colon cmt
)
```

## Long List Format Style

Long lists are lists of formal arguments in `defun`, `lambda`, or quoted lists containing more than five elements. The Long List format style applies to lists that do not fit on a single line (within the Right Text Margin).

If the Long List format style options do not appear in your Format Options dialog box, press the More Options button to display additional formatting options. The available modes for Long List format are listed below and illustrated with an example based on the following list elements, and with Right Text Margin set to 45:

```
('("entdel" "entmake" "entmod" "entnext"
"entsel" "entup"))
```

### Single-Column formatting:

```
' ("entdel "  
  "entmake"  
  "entmod"  
  "entnext"  
  "entsel "  
  "entupd"  
)
```

### Two-Column formatting:

```
' ("entdel "      "entmake"  
  "entmod"      "entsel "  
  "entnext"     "entupd"  
)
```

### Multi-Column formatting:

```
' ("entdel "      "entmake"      "entmod"  
  "entsel "      "entnext"      "entupd"  
)
```

**Fill-the-String formatting** (places as many quoted strings on one line as possible, up to the right margin):

```
' ("entdel " "entmake" "entmod" "entsel " "entnext" "entupd"  
)
```

The **Preserve Existing Line Breaks** option, if selected, may supercede the formatting indicated by Long List format style.

## Setting Case for Symbols

By default, the VLISP formatter does not change the case of AutoLISP symbols. You can set the formatter to change the case of symbols according to the VLISP protection state for symbols.

The Protected options subgroup controls the case conversion of protected symbols (built-in symbols or symbols with the ASSI GN-PROTECT flag set). The Unprotected options subgroup controls case conversion of unprotected (user) AutoLISP symbols.

### Case settings for symbols

Setting	Effect
None	Does not change the case
downcase	Forces all characters in a symbol's name to lowercase
UPCASE	Forces all characters in a symbol's name to uppercase

# Applying Visual LISP Comment Styles

The VLISP formatter recognizes five types of AutoLISP comments, and positions each comment according to its type.

Visual LISP comment formatting	
Comment	Formatted appearance
<code>;;  Inline  ;</code>	The single-line comment appears after formatting as any other expression; the multiple-line comment appears starting at a new line.
<code>;; Single-Semicolon</code>	Starts at the comment-column position, as defined by the “Single-Semicolon comment indentation” format option.
<code>;; Current-Column</code>	The comment appears starting on a new line, indented at the same level as the last line of program code.
<code>;;; Heading or 0-Column</code>	Appears on a new line, without indentation.
<code>;;_ Function-Closing</code>	Appears just after the previous expression.

The following example demonstrates each comment style.

Initial text:

```
(defun foo (x)
  ;|inline comment |;
  (list 1 2 3) ;comment-column comment
  ;;current-column comment
  ;;; heading or 0-column comment
  ) ;_function-closing comment
```

Formatted text:

```
(defun foo (x) ;|inline comment |;
  (list 1 2 3) ;comment-column comment
  ;;current-column comment
  ;;; heading or 0-column comment
  ) ;_function-closing comment
```

## Saving and Restoring Formatting Options

To save your formatting options so that they carry over to subsequent VLISP sessions, choose **Tools ► Save Settings** from the VLISP menu. Alternatively, you can save the current settings specifically for the program in the active

text editor window. VLISP saves formatter settings in a program when the Save Formatting Options in Source File option is selected. To select or cancel this option, choose Environment Options ► Visual LISP Format Options from the Tools menu. If the option is in effect, VLISP adds formatting information as comments at the end of the program, when you run the formatter.

Each formatter invocation checks for formatting options settings at the bottom of the selected text. If found, these settings override the session settings listed in Tools ► Environment Options ► Visual LISP Format Options.

## Formatter Restrictions

The following restrictions apply to the VLISP code formatter:

- The formatter relies on a fixed window font and a particular tab size. To change font settings, choose Window Attributes ► Font; to change tab settings, choose Window Attributes ► Configure Current.
- The formatter is available only within VLISP text editor windows.
- Existing SPACE and TAB characters placed outside of inline comments and strings will not influence the formatting result.

## Formatting Shortcut Keys

Press CTRL+E while in an active VLISP text editor window to display a list containing the following editor options.

Text editor code formatting commands	
Option	Effect
Indent Block	Indents the selected block of text by adding a tab to the beginning of each line.
Unindent	Unindents the selected block of text by removing a tab.
Indent to Current Level	Indents the current line to the same level as the previous line of program code.
Prefix With	Adds a text string to the beginning of the current line, or to each line in a block of selected lines, after prompting you for the string.
Append With	Appends a text string to selected lines of text, after prompting you for the string.
Comment Block	Converts a block of code to comments.

### Text editor code formatting commands (*continued*)

Option	Effect
Uncomment Block	Changes a block of comments to active text.
Save Block As	Copies selected text to a new file.
Uppcase	Converts the selected text to all uppercase.
Downcase	Converts the selected text to all lowercase.
Capitalize	Capitalizes the first letter of each word in the selected text.
Insert date	Inserts the current date (default format is MM/DD/YY).
Insert time	Inserts the current time (default format is HH:MM:SS).
Format Date/Time	Changes the date and time format.
Sort Block	Sorts the selected block of code in alphabetical order.
Insert File	Inserts the contents of a text file into the current editor window at the cursor position.
Delete to EOL	Erases everything from the cursor position to the end of the current line.
Delete Blanks	Deletes all blank spaces from the cursor position to the first non-blank character in the line.



# Checking for Syntax Errors

One main attraction of using VLISP is the extensive debugging tools it provides. These tools allow you to watch what your program is doing while it is executing, and to take a “snapshot” of your program at any point. However, VLISP also provides a number of features designed to detect program errors before you run the program.

## Checking the Balance of Parentheses

AutoLISP uses parentheses more frequently than most other computer languages. One of the most frequent syntax errors in AutoLISP is an unequal number of open and close parentheses. VLISP includes a number of tools to help you detect unbalanced or unmatched parentheses.

As noted earlier in this chapter (“To format text in an active editor window” on page 50), the VLISP code formatter searches for unbalanced parentheses when it formats your code. If you allow it to, the formatter will add parentheses where it thinks they are missing. Typically, though, the VLISP formatter adds parentheses at the end of a program, not to where you really need them. If you let VLISP add the parentheses, you will probably have to remove them later.

---

**NOTE** If you do not allow the formatter to add the balancing parentheses, it won’t format your code either!

---

In any event, you must check the structure of your program to determine where the parentheses are really missing. You can use these Parentheses Matching items from the Edit menu to help you find unbalanced parentheses:

Match Forward (CTRL +])	Moves the insertion point (marked by the cursor) just past the close parenthesis that matches an open parenthesis.  If the current cursor position is just before an open parenthesis, VLISP matches that parenthesis with its closing parenthesis. If the cursor position is in the middle of an expression, VLISP matches the current expression’s open parenthesis with its closing parenthesis.
----------------------------	---

Match Backward (CTRL + <b>]</b> )	<p>Moves the insertion point to just before the open parenthesis that matches a close parenthesis.</p> <p>If the current cursor position is just after a close parenthesis, VLISP matches that parenthesis with its opening parenthesis. If the cursor position is in the middle of an expression, VLISP matches the current expression's close parenthesis with its open parenthesis.</p>
Select Forward (CTRL + SHIFT + <b>]</b> )	<p>Moves the insertion point as the Match Forward command does, but also selects all text between the start and end positions.</p> <p>With the cursor positioned right before an open parenthesis, double-clicking also selects all text up to the matching close parenthesis, but does not move the insertion point.</p>
Select Backward (CTRL + SHIFT + <b>[</b> )	<p>Moves the insertion point as the Match Backward command does, but also selects all text between the start and end positions.</p> <p>With the cursor positioned right after a close parenthesis, double-clicking also selects all text up to the matching open parenthesis, but does not move the insertion point.</p>

For example, look at the following code:

```

1 (defun yinyang (/ origin radius i -radius half-r origin-x origin-y)
2   (setq half-r (/ radius 2))
3   (setq origin-x (car origin))
4   (setq origin-y (cadr origin))
5   (command "_ . CIRCLE"
6     origin
7     radius
8     (command "_ . ARC"
9       "_ C"
10      (list origin-x (+ origin-y half-r))
11      (list origin-x (+ origin-y radius))
12      origin
13    )
14    (command "_ . ARC"
15      "_ C"
16      (list origin-x (- origin-y half-r))
17      (list origin-x (- origin-y radius))
18      origin
19    )
20 )

```

(The line numbers are not part of the text; they are used to help explain the example.)

Here is what happens if you load this code in VLISP and continually issue the Match Forward command, starting with the insertion point at the beginning of line 1.

- VLISP does not find a matching close parenthesis, so the cursor does not move.
- Move the cursor to the beginning of line 2.
- Cursor moves to the end of line 2.
- Cursor moves to the end of line 3.
- Cursor moves to the end of line 4.
- Cursor jumps to the last right parenthesis in the program! (20)

In other words, the close parenthesis that matches the open parenthesis on line 5 is the last parenthesis in the program. You know this is an error because the last close parenthesis in an AutoLISP program should match the open parenthesis of the program's defun. Notice also that all the statements after line 5 are indented in a manner unlike in the preceding program code. These two clues indicate something is amiss at this point in the program. In fact, the close parenthesis to the command that begins on line 5 is missing.

## Using Color Coding to Detect Syntax Errors

The AutoCAD *Sample\VisualLISP* directory contains a file named *drawline-with-errors.lsp*. It is similar to the *drawline.lsp* program file introduced earlier in this manual, but it contains a couple of errors. Open the file in VLISP, so that you can see how color is used in the file:

```
(defun drawline(/ pt1 pt2) ; Local variables declared
  ;; get two points from the user
  (setq pt1 (getpoint "\nEnter the start point for the line: "))
  (setq pt2 (getpoint pt1 "\nEnter the end point for the line: "))
  ;; check to see that the two points exist
  (iff (and pt1 pt2)
    (command ".line" pt1 pt2 "")
    (princ "\nInvalid or missing points!")
    (princ) ;; exit quietly
  )
)
```

(This example also uses different fonts to represent different colors, so you can differentiate between the fonts on a black and white printed page.)

If you use the standard VLISP syntactic colorations, systems functions such as `setq`, `defun`, `getdist`, `getpoint`, and `/` are displayed in blue. The items VLISP does not recognize, such as user-defined variables, are printed in black. In this example, if you look at the unrecognized elements in the program, the word *iff* might easily catch your eye. Change it to the correct spelling, *if*, and the color immediately changes to blue.

## Using the Check Command to Look for Syntax Errors

You can perform additional syntax checking with the VLISP Check command. The Check command can detect the following errors:

- Incorrect number of arguments supplied to a known function
- Invalid variable name passed to a function (for example, a quoted symbol where a variable is required)
- Incorrect syntax in special form function calls (for example, `lambda`, `setq`, and `foreach`)

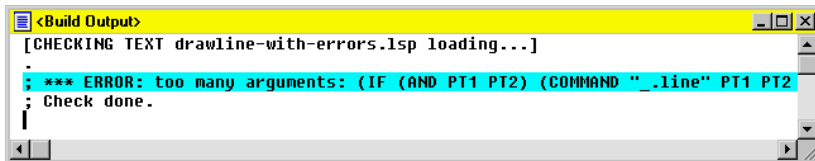
Some syntax errors can only be determined at runtime and Check cannot detect these errors. For example, if you call a function that expects an integer argument and you supply a string, AutoLISP does not detect this until runtime. As a result, this error will not be detected until you run your program.

To run the Check command on text in an editor window



- 1 Switch to the editor window containing the code you want to check.
- 2 To check the entire file, choose Tools ► Check Text in Editor from the VLISP menu.
- 3 To syntax-check a selected piece of code instead of the whole program, choose Tools ► Check Selection.

VLISP displays error messages in a new Build Output window, if it detects errors. For example, if you change the `iff` in *drawline-with-errors.lsp* to `if` and run Check, the following error message results:

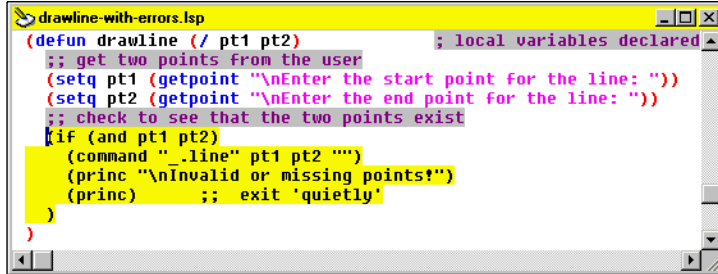


```
<Build Output>
[CHECKING TEXT drawline-with-errors.lsp loading...]
; *** ERROR: too many arguments: (IF (AND PT1 PT2) (COMMAND ".line" PT1 PT2
; Check done.
```

The message indicates that an `if` function call contains too many arguments.

## Finding the Location of the Syntax Error in Your Program

If you double-click on the error message in the Build Output window, VLISP activates the editor window, places the cursor at the beginning of the statement that caused the error, and highlights the entire expression, as follows:



```
(defun drawline (/ pt1 pt2) ; local variables declared
  ;; get two points from the user
  (setq pt1 (getpoint "\nEnter the start point for the line: "))
  (setq pt2 (getpoint "\nEnter the end point for the line: "))
  ;; check to see that the two points exist
  (if (and pt1 pt2)
    (command ".line" pt1 pt2 "")
    (princ "\nInvalid or missing points!"))
  (princ) ;; exit 'quietly'
)
```

This error results from the last `princ` statement following the `if`. The `if` statement only allows two arguments: the statement to execute if the expression is true, and the statement to execute if the expression is false. The last `princ` statement, which is used in this program to cause a quiet exit, belongs after the close parenthesis that currently follows it. (See “Exiting Quietly” on page 236 for an explanation of a quiet exit.) If you move the statement to the correct location and run Check again, the code should pass as error-free.



# Debugging Programs

Programs do not always behave in the way they were intended. When the results you get appear to be wrong, or cause the program to crash, it can be difficult to determine what is going wrong. VLISP provides many features that help you with the debugging process—finding and resolving program problems.

## In This Chapter

# 3

- Introducing Visual LISP Debugging Features
- Learning by Example
- Using the Visual LISP Debugging Features
- Using Visual LISP Data Inspection Tools

# Introducing Visual LISP Debugging Features

Debugging is usually the most time-consuming stage in the development of any program. For this reason, VLISP includes a powerful debugger that provides the following features:

- Tracing of program execution
- Tracing of variable values during program execution
- Viewing the sequence in which various expressions are evaluated
- Inspecting the values of parameters used within function calls
- Interrupting program execution
- Stepping through program execution one instruction at a time
- Inspecting the stack

VLISP provides the following facilities to implement these features:

Break Loop Mode	Halts program execution at specified points, allowing you to look at and modify the value of objects during the break. Examples of AutoLISP objects are variables, symbols, functions, and expressions.
Inspect	Provides detailed information on an object in an Inspect dialog window. If the object being inspected is composed of nested objects (a list, for example), the Inspect feature allows you to inspect all the components, each one listed on its own line within the window. You can also recursively inspect any nested object until an atomic object (such as a number or a symbol) is reached.
Watch Window	Watches the values of variables during program execution. The content of the Watch window is updated automatically. This means that if the value of a variable placed in the Watch window is changed, this change will automatically be reflected in the Watch window.
Trace Stack Facility	Views the function call stack. The call stack is a mechanism by which VLISP records the sequence of functions as they are executed by your program. You can view the stack during a debugging session (when the program is in a suspended state, such as stepping through after a breakpoint), or after your program has crashed. If



viewed after your program crashes, the function call stack shows what VLISP was doing at the moment the application failed.

Trace Facility

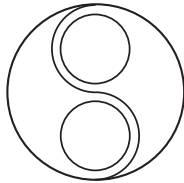
A standard LISP facility, logs the calls and returns values of traced functions into the special Trace window.

## Learning by Example

This section takes you through a VLISP sample program and demonstrates some VLISP debugging facilities along the way. You can find the sample program, *yinyang.lsp*, in the *Sample\VisualLISP* directory under the AutoCAD install directory path. Open the file in VLISP so that you can try the examples in this section.

### Stepping through the Debugging Example

First, load the file and run the yi nyang function to see what it does. The function draws the yin-yang symbol, which is used symbolically by many religions and philosophies, including Confucianism and Taoism:



When you run the program, VLISP passes control to AutoCAD and you need to respond to the prompts in the AutoCAD Command window.

VLISP evaluates AutoLISP programs by evaluating the expressions contained in parentheses. These parenthetical expressions are similar to operators in other programming languages such as C++ and Visual Basic®. The VLISP debugger uses an expression-based approach, unlike the line-by-line debuggers of languages such as C. In this approach, the debugger can suspend program execution immediately before or after the evaluation of any expression.

Debugging options are controlled from several different places within VLISP, including the text editor, the System Console, and various menus.

## Setting a Breakpoint to Interrupt Program Execution

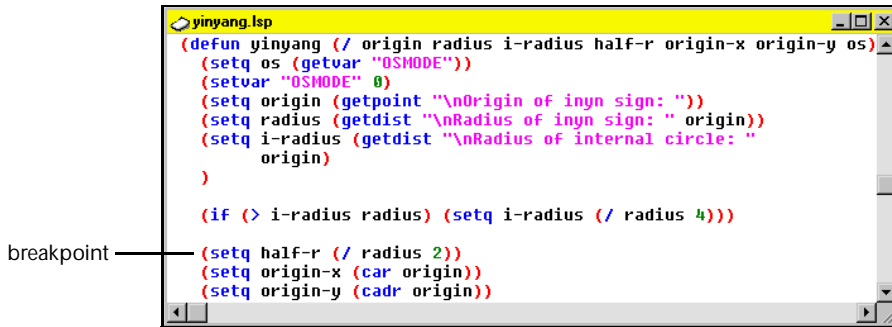
Begin by entering some debugging information in the text editor window containing the *yinyang.lsp* program.

To set a breakpoint that interrupts program execution

- 1 Move the cursor in front of the open parenthesis in the line of code that reads:

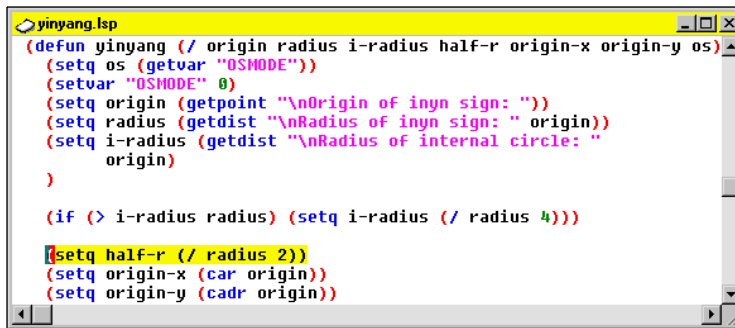
```
(setq half-r (/ radius 2))
```

The following screen snapshot indicates the position of this statement within the program:



- 2 Click the Toggle Breakpoint button in the Debug toolbar, or choose Debug ► Toggle Breakpoint from the VLISP menu. Toggle Breakpoint switches breakpoints on and off. When no breakpoint exists, Toggle Breakpoint adds a break; if a breakpoint already exists at the cursor position, Toggle Breakpoint removes it.
- 3 Load the *yi nyang* function, if you have not done so already, then run it from the VLISP Console prompt by entering the following command:  
(*yi nyang*)

After you reply to the prompts the program displays at the AutoCAD command line, VLISP halts *yi nyang* execution at the breakpoint you set and displays the code in the text editor window:



```
(defun yinyang (/ origin radius i-radius half-r origin-x origin-y os)
  (setq os (getvar "OSMODE"))
  (setvar "OSMODE" 0)
  (setq origin (getpoint "\nOrigin of inyn sign: "))
  (setq radius (getdist "\nRadius of inyn sign: " origin))
  (setq i-radius (getdist "\nRadius of internal circle: "
    origin))
  )

  (if (> i-radius radius) (setq i-radius (/ radius 4)))

  (setq half-r (/ radius 2))
  (setq origin-x (car origin))
  (setq origin-y (cadr origin))
)
```

Note how the statement following the cursor is highlighted.

## Stepping through the Program

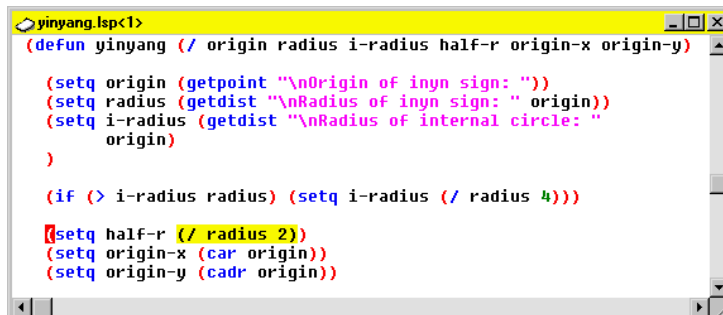
The Step commands allow you to move through a program by executing one or more expressions at a time.

To step through a program from a breakpoint



- 1 Click the Step Into button, or choose Debug ► Step Into from the VLISP menu. You can also press F8 to issue the Step Into command.

Execution begins and halts before evaluation of the inner parenthetical expression, that is, before the specified division occurs. The expression is highlighted, as shown in the following figure:



```
(defun yinyang (/ origin radius i-radius half-r origin-x origin-y)
  (setq origin (getpoint "\nOrigin of inyn sign: "))
  (setq radius (getdist "\nRadius of inyn sign: " origin))
  (setq i-radius (getdist "\nRadius of internal circle: "
    origin))
  )

  (if (> i-radius radius) (setq i-radius (/ radius 4)))

  (setq half-r (/ radius 2))
  (setq origin-x (car origin))
  (setq origin-y (cadr origin))
)
```



Now look at the Step Indicator button on the Debug toolbar; it is the last button on that toolbar.

The Step Indicator button is active when you are stepping through a program. It indicates where you are in relation to the expression at the breakpoint. The current symbol indicates that you are stopped just before an open parenthesis.



- 2 Click the Step Into button again. The cursor moves to a position directly after the evaluated expression, and the Step Indicator button indicates this.
- 3 Click the Step Into button again. The cursor moves to the end of the entire statement (the expression and all nested expressions).
- 4 Click the Step Into button again and the cursor moves to a position just before the beginning of the statement on the next line:

```

yinyang.lsp
(defun yinyang (/ origin radius i-radius half-r origin-x origin-y os)
  (setq os (getvar "OSMODE"))
  (setvar "OSMODE" 0)
  (setq origin (getpoint "\nOrigin of inyn sign: "))
  (setq radius (getdist "\nRadius of inyn sign: " origin))
  (setq i-radius (getdist "\nRadius of internal circle: "
    origin))
  )
  (if (> i-radius radius) (setq i-radius (/ radius 4)))
  (setq half-r (/ radius 2))
  (setq origin-x (car origin))
  (setq origin-y (cadr origin))
)
  
```



- 5 Now take a bigger step. Click the Step Over button, or choose Debug ► Step Over from the menu, or press SHIFT + F8 to issue this command:

```

yinyang.lsp
(defun yinyang (/ origin radius i-radius half-r origin-x origin-y os)
  (setq os (getvar "OSMODE"))
  (setvar "OSMODE" 0)
  (setq origin (getpoint "\nOrigin of inyn sign: "))
  (setq radius (getdist "\nRadius of inyn sign: " origin))
  (setq i-radius (getdist "\nRadius of internal circle: "
    origin))
  )
  (if (> i-radius radius) (setq i-radius (/ radius 4)))
  (setq half-r (/ radius 2))
  (setq origin-x (car origin))
  (setq origin-y (cadr origin))
)
  
```

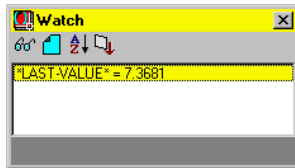
With the Step Over command, VLISP evaluates an entire expression (and all nested expressions), then stops at the end of the overall expression. The cursor moves to the end of the evaluated expression.

## Monitoring the Evaluation Results of an Expression

As you step through a program, you may want to monitor the values resulting from the evaluation of individual expressions.

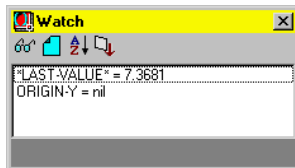
To monitor variables during program execution

- 1 From the Debug menu, choose Watch Last Evaluation.



VLISP displays the Watch window, which shows the value of the `*LAST-VALUE*` system variable. VLISP always stores the value of the last evaluated expression in the `*LAST-VALUE*` variable.

- 2 In the text editor window containing *yinyang.lsp*, double-click on any occurrence of the variable name `origi n-y`.
- 3 Click the Add Watch button in the Watch window. VLISP passes the `origi n-y` variable name to the Watch window and displays the current value of the variable in the window:



If the Watch window is not already open and you want to view a variable's value, you can open the window by choosing View ► Watch Window from the VLISP menu.

If you click the Watch window's Add Watch button without double-clicking on a variable name first, the following window appears:



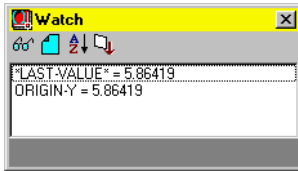
In this window, you can enter the name of the variable you want to view. VLISP may anticipate your choice by copying the name of the variable near-

est the cursor into the window. If this is not the one you want to view, simply type over the name.

VLISP updates the variables in the Watch window after each execution step.

- 4 Click the Step Over button (or press SHIFT + F8) twice.

In the Watch window, note how the value of `ORIGIN-Y` changes. It was nil at first, but after execution it took on the value corresponding to the point you clicked in the AutoCAD window.



## Continuing Program Execution



To continue running your program to the next breakpoint, or to the end, if there are no more breakpoints, press the Continue button on the Debug toolbar, or choose Debug ► Continue from the VLISP menu.

## Running in Animate Mode

Another debugging feature, animation, allows you to watch as VLISP steps through your program code and evaluates each expression. In Animate mode, it's as if VLISP repeatedly enters a Step Into command for you. Text editor windows highlight expressions being evaluated, and the Watch window continuously updates its data.

To see how Animate mode works

- 1 Turn on Animate mode by choosing Debug ► Animate from the VLISP menu.
- 2 Enter (yinyang) at the Console prompt to begin executing the program. You'll see each function highlighted as VLISP evaluates the function. You'll be prompted for input, as usual. Notice how the Watch window is updated whenever a watched variable changes. Because you previously set a breakpoint in the program, execution will halt at that breakpoint.
- 3 After you stop at the breakpoint, press the Continue button to resume execution; VLISP resumes executing in Animate mode.

You can also interrupt animation by pressing BREAK (it's the key next to SCROLL—LOCK on most keyboards). Once animation is paused you can add Watch values, set variables to new values, and add breakpoints.

To adjust the rate of animation, choose Tools ► Environment Options ► General Options, and select the Diagnostic tab. The Animation Delay setting defines the pause between program steps, in milliseconds.

To turn off Animate mode, choose Debug ► Animate from the VLISP menu again.

## Using the Visual LISP Debugging Features

In addition to setting breakpoints and running in Animate mode, as described in the “Learning by Example” section of this chapter, VLISP provides a number of other options for controlling program execution. For example:

Break on Error	Causes VLISP to break unconditionally when it evaluates the very first LISP expression encountered. You turn on this mode of operation by choosing Debug ► Break on Error from the VLISP menu.
Stop Once	<p>Automatically activates the interactive break loop whenever your program encounters a runtime error. You turn on this mode of operation by choosing Debug ► Stop Once from the VLISP menu.</p> <p>Note that if this option is selected, some errors that result from function calls entered at the AutoCAD Command prompt will cause VLISP to get focus. That is, the active window may switch from AutoCAD to the VLISP Console window, where you will be in a break loop.</p>
Break on Function Entry	Sets the Debug-on-Entry flag for a function’s name symbol, causing a break to occur every time you invoke that function. At the break, the source code for the function will be shown in a special window. You can set or clear the Debug-on-Entry flag interactively with the Symbol Service dialog box. See “Using the Symbol Service Dialog Box” on page 92 for information on setting this flag.
Top-Level Debugging Mode	Controls the loading of a program from a file or an editor window. If the option is set, breaks occur before evaluating every top-level expression (such as defun). The Top-Level debugging mode is turned on by switching off the Do Not Debug Top Level option. To find the check box

for this option, choose Tools ► Environment Options ► General Options from the VLISP menu, then click the Diagnostic tab.

If Top-Level debugging and Stop Once mode are turned on, VLISP will enter the debugging mode every time you load a file. This is due to the fact that VLISP is debugging `defun`, `setq`, and other functions defined within the file as they are loaded. This is usually not a helpful debugging technique and should only be required in rare instances.

## Starting a Debugging Session

The easiest way to start debugging is to choose Debug ► Stop Once from the VLISP menu. When this item is selected, the evaluation of the first LISP expression will be interrupted. After that you can resume program execution using various Debugger commands. Another way to enter into the debugger mode is to set a breakpoint, as shown in “Setting a Breakpoint to Interrupt Program Execution” on page 70.

When a break occurs, the corresponding VLISP text editor window will show the current LISP expression at the point which the break took place. A break loop marker will appear in the Console window. Using the Console window, you can access and manipulate the program environment in which the break occurred. You can also examine variables using the Watch window.

## Understanding Break Loops

Expressions are the basic structural units of AutoLISP, and VLISP works by repeatedly reading, evaluating, and printing expressions. In LISP terminology, this is a read-eval-print loop.

When you are running an AutoLISP program without any debugging intervention by VLISP, you are running in the Top-Level read-eval-print loop. When you evaluate an expression within the VLISP Console window, and the normal prompt is displayed, you are also working at the Top Level.

When a program's evaluation is interrupted or suspended in the middle of execution, VLISP passes control to the Console and you enter a break loop. This break loop is a separate read-eval-print loop, and is nested underneath the original read-eval-print loop. It is possible to interrupt a break loop and start yet another read-eval-print loop beneath it. The nesting level of a break loop with respect to the Top Level is called the break level.

When you enter a break loop, VLISP prefixes the Console prompt with a number indicating the level where you are located. For example, when you



first enter a break loop in a program, the prompt indicates this with the number 1:

```
_1_$
```

While you are in a break loop, you cannot switch control to the AutoCAD window.

On exiting from a break loop (for example, after issuing the Quit command), the current read-eval-print loop is terminated and the previous level loop is resumed. If you change the value of a variable in the break loop, this value will be used when the program resumes execution.

## Continuable Break Loops

There are continuable and noncontinuable break loops in VLISP. You can enter the continuable break loop at the very first break in program execution by any of the following methods:

- Turning on the Stop Once mode and reaching an expression with debugging information (that is, an expression that is loaded from source code, as opposed to from a compiled .exe file)
- Reaching a function marked for Debug on Entry
- Reaching a breakpoint you set in the program
- Entering a break loop by pressing the Pause button
- Proceeding with a Step Over, Step Into, or Step Out command from the previous break loop state

When the program is interrupted, you enter the break loop. This is apparent if the VLISP Console window is active, because the prompt is changed to reflect the current level of the break loop. In this suspended state, you have read-write access to all variables in the environment in which the break occurred. For example, if the break occurred within a function containing several local variable declarations, those variables are accessible and you can change their values by issuing setq assignments at the Console prompt.

When stopped at a breakpoint, you can control subsequent program execution by choosing one of the following items from the Debug menu, or by pressing the equivalent toolbar button:



- **Reset to Top Level** terminates all currently active break loops and returns to the Console top-level (the top read-eval-print loop).
- **Quit Current Level** terminates the current break loop and returns to a break loop one level up. This may be another break loop or the top-level read-eval-print loop.
- **Continue** resumes normal program execution from the breakpoint.

The Step commands evaluate portions of program code before resuming suspended mode:



- **Step Over** looks for the close parenthesis matching the open parenthesis where the program is currently paused, and evaluates the expressions in between.



- **Step Into** jumps into a nested expression, if any. If there are no nested expressions, it jumps to the next expression in sequence.



- **Step Out** searches for the end of the function where the program is currently paused, and evaluates all the expressions up to that point.

After exiting the break loop to the Console top-level, the Console prompt returns to its original form (without a number prefix).

## Non-Continuable Break Loops

A non-continuable break loop is activated when an error causes program interruption and the Break on Error option is set. In a non-continuable break loop, you can access all variables in the error environment, but you cannot continue program execution or execute any of the Step commands. To distinguish between continuable and non-continuable break loops, check to see if the Step and Continue toolbar buttons are active.

To leave a non-continuable break loop step, use either the Reset to Top-Level command to jump to the Console top-level loop, or Quit Current Level to return to the previous break loop level.

---

**NOTE** If you activate AutoCAD while in the midst of a non-continuable break loop, you will not be able to enter anything in the command window; in fact, the window will not contain a Command prompt. However, if you accidentally try typing anything in the AutoCAD command window, your keyboard input will be queued until AutoCAD regains control (that is, after you exit the break loop and activate the AutoCAD window). At that point, anything you typed is evaluated by AutoCAD as if you had just entered it at the Command prompt.

---

## Using Breakpoints

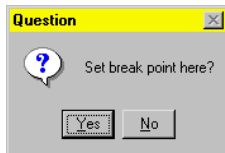
Breakpoints allow you to mark a position in a program at which program execution should be interrupted. You can set breaks to occur before or after parenthetical expressions. Breakpoints can only be set from a VLISP text editor window.

## To set a breakpoint



- 1 Move the cursor to the position at which you want to halt execution. For example, to halt execution just before the open parenthesis of an expression, place the cursor just to the left of that open parenthesis.
- 2 Press the Toggle Breakpoint toolbar button or press F9 to set the breakpoint. (For variety, you can set a breakpoint by choosing Debug ► Toggle Breakpoint from the VLISP menu, or by right-clicking the mouse and selecting Toggle Breakpoint from the resulting shortcut menu.)

If you move the cursor to an ambiguous position, such as in the middle of an expression, VLISP will move the cursor to the nearest parenthesis and display the following message asking whether you agree with the breakpoint placement:



- 3 Click Yes to accept the breakpoint location, or No if that is not where you want to set the break.

## To remove a breakpoint

- 1 Position your cursor at the breakpoint you want to remove.
- 2 Press the Toggle Breakpoint toolbar button, or press F9.

The Toggle Breakpoint works as an on/off switch. When no breakpoint exists, Toggle Breakpoint adds a break; if a breakpoint already exists at the cursor position, Toggle Breakpoint removes it. You can also use the Breakpoint Service dialog to remove breakpoints; see “Listing and Viewing the Breakpoints in Your Program” on page 80 for information on this procedure.

- 3 To remove all the breakpoints you have set, choose Debug ► Clear All Breakpoints from the VLISP menu.

## Changing Breakpoint Highlight Colors

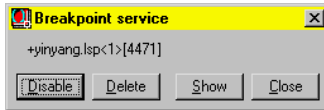
VLISP marks each breakpoint position with a colored rectangle, so you can easily locate the breakpoints in your program. By default, active breakpoints are marked in red. You can change this color by setting the :BPT-ACTIVE option in Tools ► Window Attributes ► Configure Current. See “Configure Current” on page 473 for more information on changing colors in VLISP windows.

## Disabling Breakpoints Temporarily

When using multiple breakpoints within a source file, it may be useful to disable one or more breakpoints temporarily, but leave the breakpoint position defined for possible later use. This saves time over deleting and restoring the breakpoint.

To disable a breakpoint

- 1 Place the cursor at the breakpoint marker and press the right mouse button.
- 2 From the resulting menu, choose Breakpoint Service. VLISP displays the following dialog box:

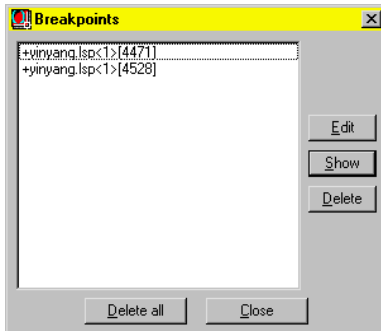


- 3 Click the Disable button in the Breakpoint Service dialog box to disable the breakpoint temporarily.

VLISP changes the color of the breakpoint marker when it disables the breakpoint. By default, it marks disabled breakpoints in blue. You can change this color by resetting the : BPT-DI SABLE option.

## Listing and Viewing the Breakpoints in Your Program

From the View menu, choose the Breakpoints window to see a list of all breakpoints currently defined to VLISP:



The Breakpoints window lists the breakpoints in all programs you are editing in VLISP, not just the program in the active editor window. In the example above, only one program (yinyang) contains breakpoints. But you could have breakpoints set in any number of files.

Each entry in the Breakpoints window shows the name of the source file containing the breakpoint, and the location of the breakpoint in the source. A leading + or - sign differentiates between active and disabled breakpoints. The dialog box allows you to delete all breakpoints at once or to edit (or display) one breakpoint at a time. Press Show to display the source position of the breakpoint. The Edit button opens the Breakpoint Service dialog, from which you can disable the breakpoint.

## Life Cycle of a Breakpoint

You can set breakpoints in a program either before or after you load the program. However, if you change the text in a program after loading the program, and then add a breakpoint, the breakpoint only takes effect after you reload the code.

Breakpoints remain in effect during the VLISP editing session and will survive between sessions if you choose Save Settings from the Tools menu.

In addition to removing breakpoints using the methods previously described in this chapter, program breakpoints are automatically lost when you do any of the following:

- Delete the code fragment containing the breakpoint
- Modify the file outside the VLISP editor (for example, edit and save it with Notepad)
- Apply VLISP formatting commands to code fragments containing breakpoints

Note also that if you modify a program's code and run it without reloading it (with the Load Active Edit Window command), the program will be interrupted when a breakpoint is reached, but the exact source position will not be shown. The following dialog box indicates this situation has occurred:



To enable the proper display of a source position, you must reload the code and restart the program.

# Using Visual LISP Data Inspection Tools

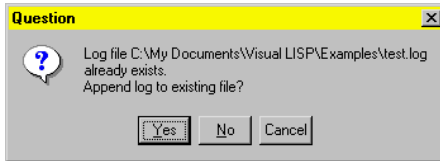
VLISP gives you almost unlimited access to symbols, values, and functions at any stage of program execution. The VLISP data inspection tools are implemented as modeless windows (except for the Symbol Service dialog box), meaning they stay on the screen as long as you need them, no matter what your program does.

- The Watch window displays the current value of any set of variables.
- The Trace Stack window displays the most current call hierarchy. At any level of the stack you can view the corresponding code, the calling code, the local variables, and more.
- The Symbol Service dialog box displays the current value of a symbol as well as its current flags. You can modify both the value and the flags from here.
- Inspect windows display any LISP object (from a string to an AutoCAD block definition) to any level of detail needed.
- Frame Binding windows display the values of all local variables for their particular stack frame (that is, the specific function invocation in the call sequence).

VLISP provides a logging feature that, when active, allows you to copy the contents of a Data Inspection window to a log file.

## To turn trace logging on and off

- 1 Activate the Trace window.
- 2 Specify a log file by choosing File ► Toggle Trace Log from the VLISP menu. Note that if the Trace window is not active, the Toggle Trace Log option will not be available.
- 3 Press Save to select the file you specified. If the file already exists, VLISP prompts you with the following message:



If you reply Yes, VLISP appends new data to the current contents of the file. If you reply No, VLISP overwrites the file and its original contents will be lost. Choose Cancel to terminate the operation and specify a different file name.

- 4 To close the log file and quit the logging process, choose Toggle Trace Log from the File menu again.

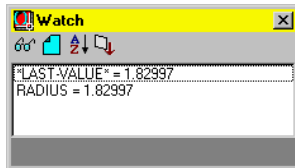
When Trace logging is turned on, any information displayed in the Trace window is also written to the log file. Most VLISP data inspection tools provide a toolbar button for copying data to the Trace window.

The state of Trace logging is indicated in the Trace window's title bar. If logging is in effect, VLISP displays the name of the log file in the title bar. If logging is off, no file name appears in the title bar.

If you do not close the log file before exiting VLISP, it closes the file automatically upon exit. After a log file is closed, you can view its contents with any text editor, such as the VLISP text editor.

## Using the Watch Window

The Watch window monitors the values of AutoLISP variables during program execution. Each Watch window element line displays the name of a variable and its current value, as illustrated in the following figure:



The Watch window is updated at each step of a VLISP interactive session and always shows the current environment state. In debugger mode, the Watch window is refreshed automatically at the end of every expression evaluation.

### To add variables to the Watch window

- 1 Highlight the variable name in any VLISP context (that is, in a text editor window, the Console window, etc.).
  - 2 Press the Add Watch button, or choose Add Watch from the Debug menu. You can also select Add Watch from a shortcut menu by right-clicking the mouse while the cursor is on a variable name.
  - 3 If the Watch window is already active, you can add variables to the watch list by clicking the Add Watch button on the toolbar in the Watch window.
- If VLISP cannot determine which variable you are interested in based on the cursor position or the text you've selected, it displays the Add Watch window:





Specify the name of the variable to be watched in this window, then click OK.

The Watch window retains its variables during a VLISP session. This means that if you invoke Watch, add variables to the Watch window, and then close the Watch window, the variables you added will appear in the Watch window, if you invoke Watch again during the current session.

The introductory section of this chapter includes an example of using the Watch window (See “Monitoring the Evaluation Results of an Expression” on page 73.)

## Using the Watch Toolbar

The toolbar on the Watch window contains the following buttons:



Add Watch

Invokes the Add Watch command to add a new variable to the Watch window. This variable can be selected from any active text window or typed in the Add Watch dialog box.



Clear Window

Removes all variables from the Watch window.



Sort  
Expressions

Sorts the variables in the Watch window alphabetically by name.



Copy to  
Trace/Log

Copies the contents of the Watch window to the Trace window. If logging is active, the contents of the Watch window are also copied to the trace log.

## Using the Watch Item Shortcut Menu

To display the Watch Item shortcut menu, select an item in the watch list and right-click.

The following items appear on the Watch Item menu:

Inspect Value

Invokes the Inspect feature for the selected value. (See “Using Inspect Windows” on page 95.)

Copy Value

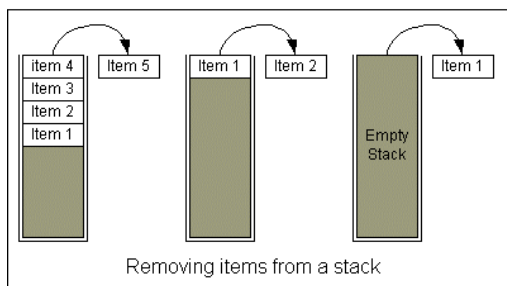
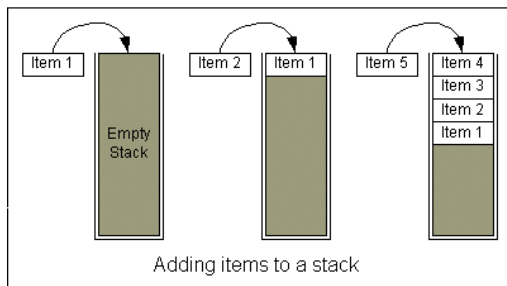
Copies the value of the selected variable into system variable \*obj\*.



Print Value	Prints the selected variable value in the Console window, prefixed with a single quote (').
Symbol	Calls the Symbol Service dialog box for the selected variable. (See “Using the Symbol Service Dialog Box” on page 92.)
Apropos	Calls the Apropos dialog box using the selected symbol’s name as the <i>Apropos</i> argument.
Remove from Watch	Removes the selected variable from the Watch window.

## Understanding the Trace Stack Window

VLISP has a special debugging tool called a trace stack which is a historical record of the execution of functions within your program. (The term *stack* is derived from a computer programming structure of the same name.) The following figure illustrates adding and removing items from a stack. You can see why a stack structure is often referred to as LIFO—Last In, First Out:



The trace stack is used by VLISP to “remember its way out” of a nested series of expressions. By viewing the stack, you can see what is happening within your program as it is executing (within a suspended break mode) or immediately after it has crashed.

Before you invoke a function at the Console window or from AutoCAD, the trace stack is empty. The action of invoking a function causes a record, or element, to be placed on the stack. As that function calls additional nested functions to perform the work of your program, additional elements may be added to the stack. VLISP only needs to place elements on the stack when it needs to remember its way out of nested functions.

There are two conditions where it is useful to examine trace stacks. The first is when your program is in a suspended state, such as during a breakpoint pause. The second is after an error occurs, causing your program to fail.

## Stack Element Lists

A stack element is an individual record or line-item history within a trace stack. There are five kinds of elements that may appear within a stack:

- Function call frames show one individual function invocation. Each function call frame appears in the following format:

*level (function-name {argument1}...)*

Arguments within this listing are displayed not by their local parameter name, but by the values that were actually passed to the function.

- Keyword frames are displayed at the very top and bottom of a trace stack. They are displayed in the following form:

*level :keyword- {optional -data}*

The *keyword* indicates the type of the frame. The *optional-data* displays additional information relating to the state of the program.

- Top forms indicate an action that was initiated by typing an expression at the top-level Console window, or from the invocation of a function that was triggered during the loading of a file or selection within a VLISP editor window.
- Lambda forms are placed within a stack whenever a lambda function is encountered within your program.
- Special forms display the invocation of the `foreach` and `repeat` functions. The arguments for these functions are not displayed. They appear as:  
*level (function-form ...)*

Function call frames and keyword frames are discussed in more detail in the following sections. These sections use the following code to demonstrate the trace stack. If you wish, you can copy this code into a VLISP editor window, set a breakpoint as indicated in the code comments, and run this sample:

```

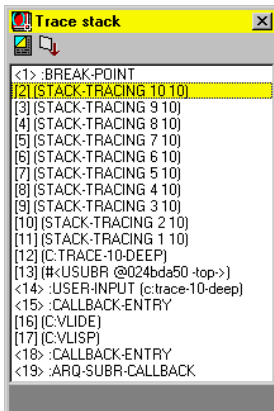
(defun stack-tracing (indexVal maxVal)
  (princ "At the top of the stack-tracing function, indexVal = ")
  (princ indexVal)
  (if (< indexVal maxVal)
      (stack-tracing (1+ indexVal) maxVal)
      (princ "Reached the maximum depth.") ; place a breakpoint
                                           ; at the beginning of
                                           ; this line
  )
)
(defun c:trace-10-deep ()
  (terpri)
  (stack-tracing 1 10)
)

```

## Viewing the Current Trace Stack



To see the state of a function call stack while your program is suspended at a breakpoint, choose View ► Trace Stack from the VLISP menu, or click the Trace toolbar button. VLISP displays the Trace Stack window:



The Trace Stack window displayed above shows a function call frame for the `stack-tracing` function. The second element, or frame, in the trace stack is highlighted:

```
[2] (STACK-TRACING 10 10)
```

The number [2] simply identifies it as the second element in the stack. The numbers following the `stack-tracing` function name (10 10) indicate the actual values that were passed to the function.

## Displaying Information on a Trace Stack Element

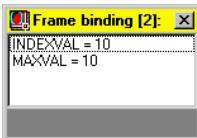
To obtain more information about an element in the trace stack, select the element and right-click to display a shortcut menu.

Active items available on the shortcut menu depend on the type of stack element you selected before right-clicking. Possible menu commands include the following:

Inspect	Calls the Inspect feature for the selected stack element.
Print	Prints the stack element to the Console window.
Function Symbol	Calls the Symbol Service feature for the function call in the stack frame, if the function is called by the symbol.
Copy	Copies the selected trace stack element to system variable *obj*.
Local Variables	Displays the Frame Bindings dialog box to allow browsing of local variable values at the time the function was called; see “Using the Frame Binding Window” on page 88.
Source Position	Checks whether or not the source text is available for the function called at the selected stack frame. If the source code is available, the text window with the source code is displayed, with the current position inside the function highlighted.
Call Point Source	Shows the position of the caller expression, similar to Source Position.

## Using the Frame Binding Window

Choose the local variables item from the Trace Stack shortcut menu to display the Frame Binding window:



The Frame Binding window displays information about the local variables in the frame. In the example shown above, it lists the parameter names (INDEXVAL, MAXVAL), along with the values assigned to these parameters. These values were passed to the function. The parameters are listed in the order they are defined within the function.

If you right-click on an entry in the Frame Binding window, VLISP displays a shortcut menu containing the following items:

Inspect	Calls the Inspect feature for the selected value.
Print	Displays the selected value in the Console window.

Symbol	Calls the Symbol Service dialog box for the selected symbol.
Copy	Copies the selected value into the system variable *obj*.
Add to Watch	Adds the selected symbol to a Watch window.

## Understanding Keyword Frames

A keyword frame indicates a specific type of operation that occurs within the VLISP environment. The keyword indicates the type of operation. Keyword frames will appear in only two locations: at the very top of the stack, or at the very bottom of the stack.

The following types of keyword frames will appear only at the bottom of a stack:

Keyword frames (bottom)	
Frame type	Operation that occurred
:ACAD-REQUEST	A call to the function shown in the frame immediately above the :ACAD-REQUEST keyword frame was invoked from the AutoCAD command prompt.
:DCL-ACTION	Execution of a DCL tile or dialog action was requested from AutoCAD. The keyword :DCL-ACTION is followed by two strings: the DCL dialog name and the value of the \$KEY variable in the DCL action body. If a number appears, it is the value of the \$REASON variable in the DCL action body. The frame immediately above the keyword describes the function call built from the action string.
:INSPECT-EVAL	Evaluation of an Inspect command.
:INSPECT-VERBOSE	Entrance into a drawing Inspect hook function.
:TOP-COMMAND	The VLISP IDE requested the action resulting in the first element placed within the stack. This situation occurs, for example, when a function is invoked directly from loading a selection or a file.
:USER-INPUT	The character string shown in the frame was entered from the VLISP Console window. The frame immediately above the keyword describes the expression as it was translated from the user input. If the input string is too long, right-click to open a context menu, and choose Show Message to view the entire text. You can also choose the Inspect command to inspect the entered string.
:WATCH-EVAL	Evaluation of a watch expression.

The following types of keyword frames may appear at the top of a stack:

Keyword frames (top)	
Frame type	Operation that occurred
:ACMD-CALLBACK	Registered AutoCAD command call.
:AFTER-EXP	Indicates that your program is interrupted in a debugging break mode, and the Step Into or Step Over command just stepped out of an expression.
:ARQ-SUBR-CALLBACK	Indicates a normal call from AutoCAD to a VLISP-defined function.
:AXVLO-IO-CALLBACK :DWF :DWG	Saves or restores a VL object in a DWG.
:BEFORE-EXP	Debugger break upon entering the function. This message will appear whenever you are stepping through using Step Into or Step Over, and the step is entering an expression (as opposed to just leaving an expression, which is indicated by the :AFTER-EXP keyword).
:BREAK-POINT	User-specified breakpoint.
:ENTRY-NAMESPACE	A call in the context of a separate-namespace VLX.
:ERROR-BREAK	General runtime error. The Show Message right-click context menu selection allows you to view more specific error messages.
:FUNCTION-ENTRY	Debugger break upon entering the function. The stack element following this message contains the call frame for the function in which the break occurred.
:KBD-BREAK	The PAUSE key was pressed, placing the program on hold.
:PROTECT-ASSIGN	Assignment of a value to a protected symbol. From the right-click shortcut menu, you can choose Show Message to view the variable name, the current value, and the new value that was attempted to be assigned to the variable. You can also choose the Inspect command to view the list containing the symbol, and the new value indicated following :PROTECT-ASSIGN.
:REACTOR-CALLBACK	Reactor call.

## Keyword frames (top) (continued)

Frame type	Operation that occurred
:READ-ERROR	Error during a read operation. The Show Message right-click context menu selection provides additional information about the error.
:SYNTAX-ERROR	VLISP encountered incorrect AutoLISP program syntax.

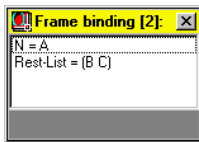
## Understanding Special Function Call Frames

There are two special function call frames.

The FOREACH frame indicates a call to the `foreach` function. From the short-cut menu, choose the Local Variables option to display the name and current value of the user-supplied variable and list variables bound by the `foreach` function. For example, if the following expression were evaluated

```
(foreach n '(a b c) (print n))
```

then selecting the Local Variables option displays a Frame Binding window like the following:



This Frame Binding window identifies the user-supplied variable (N), the current value of that variable (A), and the items remaining to be processed in the list supplied to `foreach` (B C).

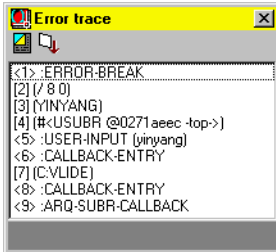
The REPEAT frame indicates a call to the `repeat` function. From the shortcut menu, the Local Variables command displays the special name *counter* and the current value of the repeat internal counter. The internal counter value is initially set to the integer value passed to `repeat`, indicating the number of iterations desired. The counter decreases by one at each loop iteration. It shows the number of iterations remaining, minus one.

Note that each repeat expression possesses its own counter, but only one such counter can be added to the Watch window.

AutoLISP functions such as `if`, `cond`, `and`, and `setq` do not appear on the stack. They are not necessary because their call position may be viewed within the source file in the VLISP text editor window.

## Viewing an Error Trace Stack

If your program terminates due to an error, choose Error Trace from the View menu to see the state of function invocations up to the time your program crashed:



The error trace is a copy of the trace stack as it appeared at the time the error occurred. If the Break on Error debugging option is selected, the error trace and the trace stack are identical immediately after an error occurs. You can see this by selecting Break on Error from the Debug menu, intentionally causing an error (for example, issuing a function call that divides by zero), and opening the two trace windows.

The toolbar on the Trace Stack window contains two buttons:



Refresh

Refreshes contents of Trace Stack window.



Copy to  
Trace/Log

Copies the window contents to the Trace Stack window or open log file.

When you issue a Reset command to exit a break loop (for example, Reset to Top Level), pressing the Refresh button in the Trace Stack window replaces that window's contents with the latest trace stack data. In contrast, refreshing the Error Trace window does not change the window's contents, unless a subsequent error has occurred.

## Using the Symbol Service Dialog Box

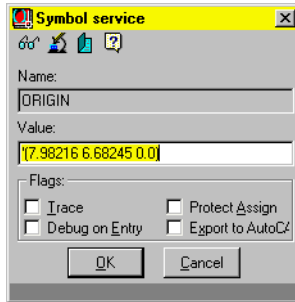
The Symbol Service feature is designed to simplify access to the different debugger features provided for symbols. Most facilities available for symbols are also available through this feature.



## To open a Symbol Service dialog box and update a symbol



- 1 Highlight the name of any symbol in your program's source code or in the Console window.
- 2 Choose View ► Symbol Service from the VLISP menu, or press the Symbol Service button on the Debug toolbar.



The Symbol Service dialog box contains the following components:

- A toolbar
  - A Name field, where you can enter or change the symbol to work on
  - A Value field that displays the symbol's value or its initial substring
  - A series of check boxes for symbol flags described in “Understanding Symbol Flags” on page 94
- 3 To update the value of the displayed symbol, enter an expression in the Value field. When you press OK, VLISP evaluates the expression and assigns its value to the symbol.

If the symbol you specified is a protected symbol, the Value field will be read-only. To remove protection, clear the Protect Assign check box. See “Understanding Symbol Flags” for more information on Protect Assign.

Use the OK and Cancel buttons to close the dialog box and to continue working in VLISP.

## Using the Symbol Service Toolbar

The Symbol Service toolbar contains the following buttons:



Watch

Adds the symbol to the Watch window.



Inspect

Opens the Inspect window to show the value of the symbol.



Show  
Definition

If the symbol names a user-defined function, this command opens the text editor window containing the function definition and highlights the function.



Help

Displays information from the VLISP Help file, if the symbol refers to a built-in function.

## Understanding Symbol Flags

The Symbol Service dialog box provides direct access to symbol flags and properties of functional objects that may be associated with them. The following symbol flag options are available:

Trace (Tr)

The Trace flag activates the tracing of any user-defined function (shown as a symbol within the Symbol Service window). Tracing will only occur when the symbol is a function, and the expression being evaluated uses the symbol name as a function (not as a local variable name, for example).

Protect Assign  
(Pa)

This flag intercepts attempts to assign values to protected symbols. For instance, the symbol *pi* is a protected symbol. All symbols that are the names of built-in AutoLISP functions are assignment-protected by default. See “Protected Symbols” on page 228 for more information on symbol protection.

Note that symbol protection works only for explicit `setq`, `set`, or `defun` invocations. Binding a protected symbol in an argument list of a user-defined function is not intercepted.

Debug on  
Entry (De)

If this flag is set, a breakpoint occurs at each function invocation, regardless of whether the function was loaded with debugging information. The `De` flag is tested at each function invocation, not during `load` or `defun` execution.

Note that VLISP ignores the Debug-on-Entry flag for all SUBR and EXRXSUBR symbols.

Export to  
ACAD (Ea)

If the Ea flag is set, the function associated with this symbol is defined as an external subroutine. This makes the function available to ObjectARX applications.

## Using Inspect Windows

The Inspect feature is the component of VLISP that provides you with the ability to browse, examine, and modify AutoLISP and AutoCAD objects. You can use Inspect to view the following items:

- Any AutoLISP objects such as lists, numbers, strings, and variables
- AutoCAD drawing entities
- AutoCAD selection sets

Using Inspect, you can also browse through complex data structures.

The Inspect tool creates a separate window for each object you inspect.

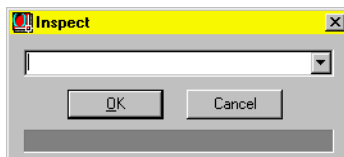
### To open an Inspect window



- 1 Select an AutoLISP object name (for example, a variable).
- 2 Choose View ► Inspect from the VLISP menu, or press the Inspect button on the Debug toolbar.

The Inspect command is also available from a number of shortcut menus and from the windows displayed by the Apropos and Symbol Service features.

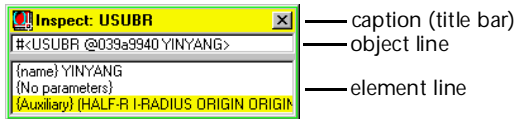
- 3 If you invoke the Inspect command without selecting an object name, VLISP prompts you to specify the object you want to inspect, displaying the following dialog box:



Enter the object or expression you want to inspect, then press OK to open the Inspect window or press Cancel to cancel the action.

VLISP saves the last 15 items you enter in the Inspect prompt box. You can choose a previously specified object for inspection by selecting it from the drop-down list.

For example, to inspect the definition of the *yi nyang* function, select the name in the text editor window containing the *yinyang.lsp*, then press the Inspect button to view the Inspect window:



## Using the Inspect Window

All Inspect windows have a similar appearance and contain a caption, an object line, and an object element list (which may be empty). The window elements contain the following content:

- The caption of an Inspect dialog box shows the type of object being inspected.
- The object line shows a printed representation of the inspected object.
- The element list displays the components of the inspected object.

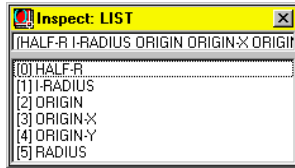
The element list may vary in size and content for different object type. Each element list is shown as a pair: name and content. The name is enclosed in brackets. Square brackets ( [ ] ) denote that you can modify the item by issuing a Modify command from the shortcut menu associated with the item, and curly brackets ( { } ) indicate that you cannot modify the item.

Both the object line and the element list lines have their own associated shortcut menus. These menus are described in “Common Inspect Commands” on page 102.

VLISP will display up to 50 element lines in an Inspect window. If there are more than 50 elements to be shown, Inspect displays the elements in a series of pages. When you scroll to the bottom of the Inspect window and there are more entries remaining to be displayed, the bottom of the list contains a ">>> [Next page]" element line. To navigate among the pages, use the following procedures:

- To page down, double-click on the ">>> [Next page]" element line, or select that line and press ALT+E.
- For Inspect windows showing AutoLISP lists and selection sets, you can page up by double-clicking the "<<< [Previous page]" element line, which appears at the top of the list. (Or select that line and press ALT+E.)
- For AutoLISP lists and selection sets, when you reach the last page of element lines, you can return to the first page by double-clicking on the "<<< [First page]" element line, or by selecting that line and pressing ALT+E.

VLISP expands an item in the element list if you double-click on it. For example, the {Auxiliary} component in the sample Inspect window is itself a list. Double-click on the {Auxiliary} item to open another Inspect window showing the elements in the list:



## Understanding Object Element List Formats

The contents of an Inspect element list vary, depending on the data type of the object being inspected. The following table identifies the list contents for each data type.

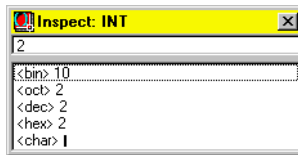
Inspect element lists	
Data type	Contents of element list
INT (integer)	The various representations of integers.
REAL (floating point number)	Empty.
STRING	The sequence of characters in the string, which may in turn be inspected as integers.
SYMBOL	Three elements: value, print name, and flags.
LIST (for proper lists)	Items of the inspected list.
LIST (for improper lists)	Two elements: the car and cdr fields. It serves for all cases that are not proper LISP lists, that is, where the last cdr is not nil.
FILE	The name of the corresponding file and the file's opening attributes.
SUBR, EXRXSUBR, and USUBR	The name of the function (the name that was specified in <b>defun</b> or at load time). SUBR refers to internal and compiled functions, EXRXSUBR refers to external ARX functions, and USUBR identifies user-defined functions.
ENAME (drawing entity)	The fields in this element list correspond to the AutoCAD DXF object list, as returned by the AutoLISP built-in function.

## Inspect element lists (*continued*)

Data type	Contents of element list
PICKSET (selection set)	List of selected AutoCAD objects.
VARIANT	The data type and value of the variant.
SAFEARRAY	The data type, number of dimensions, and value of the safearray.

Sample Inspect windows for each data type follow.

**INT**                      The INT (integer) Inspect window shows the number represented in binary, octal, decimal, hexadecimal, and character formats. Character format means the ASCII character that corresponds to the number (for large numbers it takes the last byte).

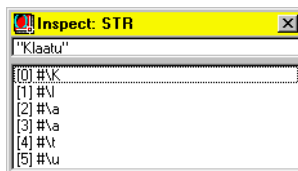


The INT Inspect window does not have an element list.

**REAL**                      The REAL Inspect window does not have an element list.



**STRING**                      Shows the string as a list of characters represented as numbers:



Double-click on a listed character to see its number representation.

## SYMBOL

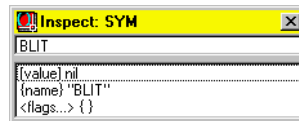
Contains the symbol name, the symbol value, and the flags that represent symbol attributes. Flags may be one of the following:

**Pa** Protect Assign

**Tr** Trace

**De** Debug on entry

**Ea** Export to ACAD

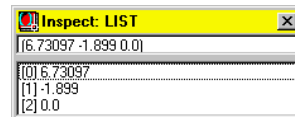


To change a symbol's value or flag settings, use the object line menu command Symbol Service, which shows the Symbol Service window.

Note that the information supplied by the SYMBOL Inspect window is available more conveniently through the Symbol Service feature.

## LIST (proper list)

Shows the elements of a proper list:



## LIST (improper list)

Shows the car and cdr of an improper list. For example, a list constructed by `(cons 4 '(5 . 0))` is represented as follows:



## FILE

File Inspect fields include the following:

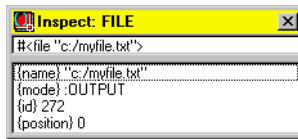
**File name** is the name string used in the open function.

**Mode** indicates whether the file is open for input, output, append, or whether the file is closed.

**ID** shows the internal file identifier.

**Position** shows the current position in the file.

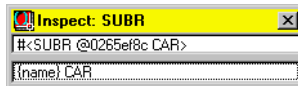
**EOF** indicates whether or not the end of the file has been reached. This field does not appear if a file is open for output.



## SUBR

The SUBR data type represents functions that cannot be debugged with the VLISP debugging tools (for example, you cannot set breakpoints). These are internal AutoLISP functions, or functions loaded from FAS or VLX files.

The SUBR Inspect window shows a string containing the name of the symbol, as in the following example:

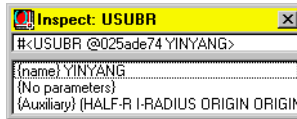


## USUBR

The USUBR data type represents functions that can be debugged with the VLISP debugging tools (for example, you can set breakpoints and view the values of program variables). These functions are loaded from LISP source code.

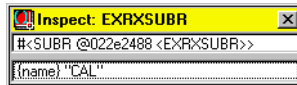
The USUBR Inspect window shows the name of the symbol, a list of function parameters (arguments), and a list of local variables declared in the function (listed after the " / " in the defun argument list). The following example shows an Inspect window for a function that accepts no arguments and declares several local variables:





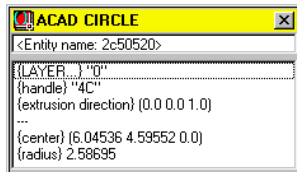
EXRXSUBR

The EXRXSUBR data type represents functions loaded from external ARX applications. The EXRXSUBR Inspect window shows a string containing the function name, as in the following example:



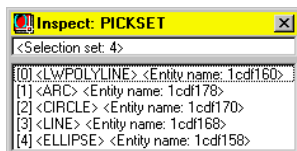
ENAME

The contents of the ENAME Inspect window depend on the properties of the entity being inspected. The following example shows an Inspect window for a circle:



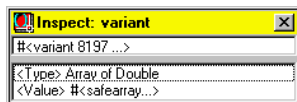
PICKSET

The PICKSET Inspect window lists the elements in a selection set:



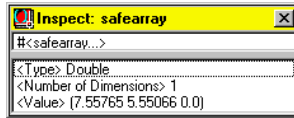
VARIANT

The VARIANT Inspect window shows the data type and value of the variant. The following example shows an Inspect window for a variant that contains an array of doubles:



## SAFEARRAY

The SAFEARRAY Inspect window shows the data type, number of dimensions, and value of the safearray. The following example shows a Safearray Inspect window for a single dimension array of doubles:



You can also use the Inspect feature to examine ActiveX objects. See “Using the Inspect Tool to View Object Properties” on page 166 for an example of this.

## Common Inspect Commands

The Inspect windows provide shortcut menus containing commands relevant to the data being inspected.

To display the object line shortcut menu, press ALT+O, or right-click the object line. The following commands may be present in an object line shortcut menu:

Symbol Service	Invokes the Symbol Service feature.
Print (ALT+P)	Prints the object in the Console window.
Pretty Print	Formats and prints the object in the Console window.
Copy	Copies the object to the *obj* variable.
Log	Copies the current contents of the Inspect dialog box to the Trace window. If logging is active, the contents are also copied to the trace log.
Update (ALT+U)	Updates the Inspect dialog box to show the most recent status of the inspected object.

The element line shortcut menu appears after highlighting the element line and right-clicking. The following commands may appear on the element line shortcut menu:

Inspect (ALT+I)	Calls Inspect and passes it the element value as an argument.
Descend (ALT+D)	Calls Inspect, passes it the element value as an argument, and closes the current Inspect window.

Copy	Copies the value of the inspected element to the <code>*obj*</code> variable.
View Source	Activates a text editor window containing the selected text. If the text was loaded from the Console window or from a list representation, this command activates a new text editor window.

The default command for an element line, invoked by pressing ENTER, is the Inspect command.

## Copying Inspect Objects to the `*obj*` System Variable

Sometimes it is useful to access some part of an object from your program or from the VLISP Console window. You may also want to copy the value of one object's item into another item and so on. To perform all these tasks, the Inspect feature manages a reserved system variable named `*obj*`. This variable can be used as a temporary storage area while browsing through data structures. From inside an Inspect dialog box, you can assign a value to this variable and replace the value of the current item with the value of `*obj*`.

To assign the value of an inspected object to the `*obj*` variable, right-click the item in the Inspect window and choose Copy.

## Handling Errors in the Inspect Command

In text editor windows it is not possible to inspect selected expressions longer than 256 characters. If you select a string longer than 256 characters, you will be prompted to enter an object name.

If you specify an object or expression that VLISP cannot evaluate, VLISP issues a standard AutoLISP error message. Once the error message appears, you can correct the expression in the dialog box and try to evaluate it once more.

Errors arising from evaluation of the object you entered cannot be investigated from a nested break loop, because all breaks are disabled during such evaluation. If you wish to examine the error, choose View ► Error Trace from the VLISP menu, or copy the expression to the Console prompt and press ENTER.

## Closing All Inspect Windows

To close all Inspect windows, choose Window ► Close Windows ► Inspectors from the VLISP menu.

# Viewing AutoCAD Drawing Entities

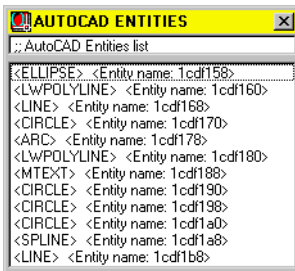
VLISP provides facilities to walk through the AutoCAD drawing database and inspect the raw data for each drawing entity reported by AutoCAD. You access drawing entities through the VLISP Browse Database feature. Browse Database displays entity information in Inspect windows. You can set a diagnostic option telling VLISP how much information to supply about entities.

To control the amount of Inspect information displayed for drawing objects

- 1 Choose Tools ► Environment Options ► General Options.
- 2 Click the Diagnostic tab in the General Options window.
- 3 Select Inspect Drawing Objects Verbosely to view detailed entity information. Clear the option check box to minimize the amount of information supplied by Inspect.

## Viewing Entities in the Drawing Database

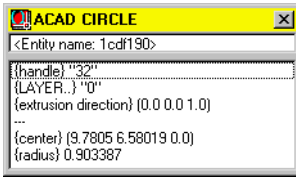
To Open an Inspect window for the collection of entities in the current drawing database, choose View ► Browse Drawing Database ► Browse All Entities from the VLISP menu. VLISP displays a window listing the entities in the database:



Note that VERTEX and ATTRIB entity types are not included in this list. You access these entity types through their parent entities, which are available when you inspect POLYLINE and INSERT entities.

The shortcut menu commands available for the object line in the ACAD Entities Inspect window are Log and Update.

To open an Inspect window for a specific entity, double-click on the entity name, or select the entity, then right-click, and choose Inspect:



The title bar of this window identifies the drawing entity type. The object line of the window displays the entity name.

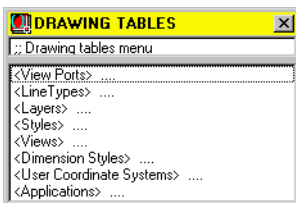
<Entity name: 1cdf190>

The shortcut menu for the object line contains the common Inspect commands Print, Copy, Log, and Update, plus some new items.

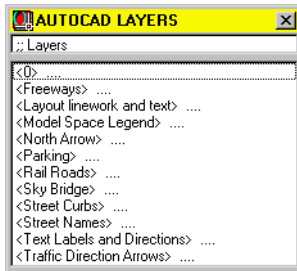
Modify	If available, this command opens the standard AutoCAD DDMODIFY dialog for the inspected entity.
Inspect Raw Data	Displays an Inspect window containing the list resulting from an entget function call for the entity.
Inspect Next Entity	Displays an Inspect window for the next entity in the entities list.
Inquire Extended Data	Displays a list of applications currently registered by regapp. If you select an item from the list, any extended data related to the chosen application is included in the Inspect entget list. See “Viewing Extended Data” on page 107 for more information.

## Viewing Symbol Tables in the Drawing Database

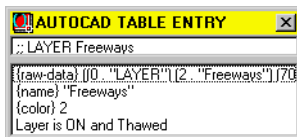
Choosing View ► Browse Drawing Database ► Browse Tables from the VLISP menu opens an Inspect window for the collection of symbol tables in your drawing:



You can inspect each table as a collection of named attributes. Double-click on a name to view its attributes, or select the name, right-click, and choose **Inspect**:

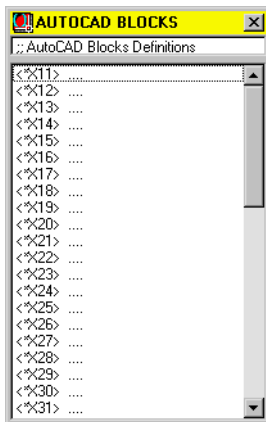


To view a table entry for a selected attribute, double-click on the attribute name, or select the attribute, right-click, and choose **Inspect**:

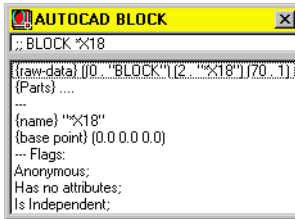


## Viewing Blocks in the Drawing Database

Choose **View ► Browse Drawing Database ► Browse Blocks** from the VLISP menu to open an **Inspect** window for the blocks in your drawing:



Double-click on the block name you are interested in to open an **Inspect** window for the block, or select the block, right-click, and choose **Inspect**.

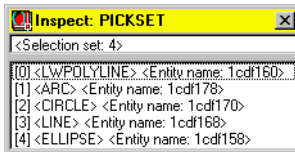


The raw-data element shows the symbol table entries for the inspected block. Double-click on the parts item to open an Inspect window listing the collection of entities residing within the block.

The raw-data and parts element lines occur in all block Inspect windows. Other element lines, such as {name}, appear only if the Inspect Drawing Objects Verbosely Diagnostic option is selected. See “Diagnostic Options” on page 477 for information on setting VLISP diagnostic options.

## Viewing Selected Objects in a Drawing

Choose View ► Browse Drawing Database ► Browse Selection from the VLISP menu to select the drawing objects you want to view. VLISP invokes the `ssget` function to prompt you to define a selection set in the AutoCAD drawing window. When you complete the selection, VLISP opens the Inspect window for your selection:



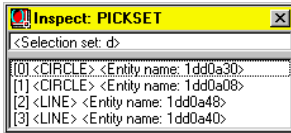
Double-click on an entity name to open an Inspect window for the entity, or select an entity, right-click, and choose Inspect.

## Viewing Extended Data

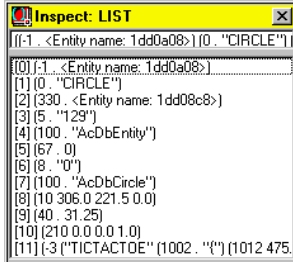
Choose View ► Browse Drawing Database ► Inquire Extended Data from the VLISP menu to see a list of the applications currently registered (through `regapp`) as containing extended data. If you select an application from this list, its extended data is included into the Inspect `entget` list.

To view extended data associated with an AutoCAD object

- 1 Choose View ► Browse Drawing Database ► Inquire Extended Data from the VLISP menu.
- 2 Select the application whose data you are interested in viewing.
- 3 In the AutoCAD window, select the drawing objects whose extended data you want to view.
- 4 From the VLISP menu, choose View ► Browse Drawing Database ► Browse Selection. VLISP displays an Inspect window listing the AutoCAD objects you selected:



- 5 In the Inspect window element list, double-click on an object whose extended data you want to view. VLISP displays an Inspect window for the object.
- 6 Select the object line in the Inspect window and right-click to view a shortcut menu.
- 7 Choose Inspect Raw Data from the shortcut menu. VLISP displays an Inspect window like the following:



Extended data is identified by the -3 DXF group code. The last line in the entity list shows the extended data for the selected object. You can double-click on this line to open a separate Inspect window containing just the extended data.



# Building Applications

This chapter describes how to build applications with VLISP.

VLISP allows you to compile your program files and create a single executable module that you can distribute to users. The first part of this chapter provides basic knowledge about the VLISP compiler and may be sufficient for building macros and small programs that work in a single document. The rest of the chapter helps you build more complex applications. These remaining sections discuss design considerations for an environment where several AutoCAD drawings may be open at the same time, and provide information on fine-tuning the performance of compiled code.

## In This Chapter

# 4

- Compiling and Linking Programs
- Making Application Modules
- Designing for a Multiple Document Environment

# Compiling and Linking Programs

Each time you load AutoLISP source code, the code is translated into instructions the computer understands (executable code). The advantage of having source code translated each time you load it is that you can make a change and immediately try it out. This is useful for quickly testing new code, and for debugging that code.

Once you are sure your program is working correctly, translating AutoLISP source code each time it loads is time-consuming. VLISP provides a compiler that generates executable machine code files from your source files. These executable files are known as FAS files. Because the executable files contain only machine-readable code, the source code you spent weeks or months developing remains hidden even if you distribute your program to thousands of users. Even strings and symbol names are encrypted by the VLISP file compiler.

VLISP also provides features for packaging more complex AutoLISP applications into VLISP executable (VLX) files. VLX files can include additional resources files, such as VBA and DCL files, and compiled AutoLISP code. See “Making Application Modules” on page 117 for instructions on building VLX files.

Using VLX files, you can further control your application’s operating environment by exposing only those functions you choose to expose, and by maintaining a wall between your program’s variables and the variables users can interact with in AutoCAD. For more information on controlling the operating environment of a VLX, see “Designing for a Multiple Document Environment” on page 126.

## Using the Compiler

VLISP provides several ways to access and use the file compiler. To compile a single AutoLISP file you can use the `vlisp-compile` function. To compile many AutoLISP files into a single VLX file, you can use the Make Application wizard. The `vlisp-compile` function and the Make Application wizard are described in this chapter.

If your application consists of a set of AutoLISP files loaded in parallel, it is recommended that you use the VLISP integrated project management facilities to compile your files. The project manager automatically recompiles files that have changed, allows you to find code segments without knowing which files contain them, and optimizes the use of function calls and local

variables in the compiled files. These features are explained in detail in “Managing Multiple LISP Files” on page 138.

## Compiling a Program from a File

To compile a single AutoLISP file, call the `vlisp-compile` function. The function syntax is

```
(vlisp-compile 'mode "filename" [out-filename])
```

For this function

- *mode* is a symbol identifying the compiler mode
- *filename* is a string naming the source file
- *out-filename* is a string naming the compiled output file

### Choosing a Compiler Mode

The mode parameter indicates the compilation mode, which can be one of the following:

<code>st</code>	Standard build mode
<code>lsm</code>	Optimize and link indirectly
<code>lsa</code>	Optimize and link directly

The standard mode produces the smallest output file and is suitable for programs consisting of a single file.

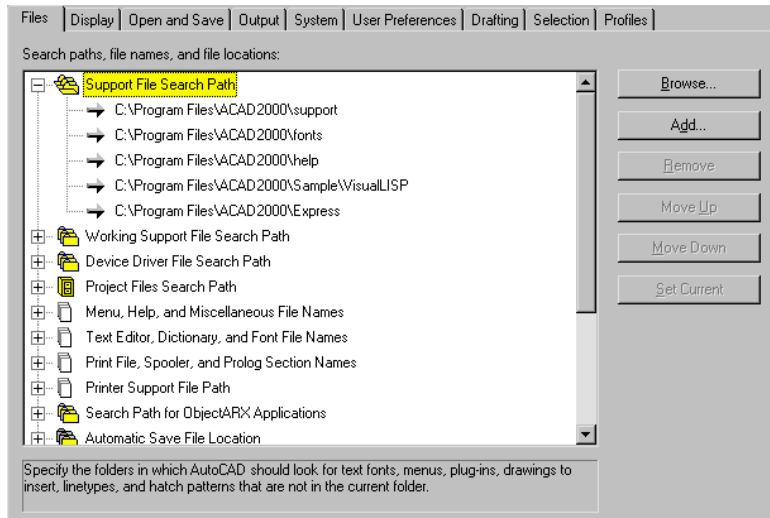
The optimization options result in more efficient compiled files, which becomes important as your programs grow in size and complexity. The basic functions of optimization are as follows:

- Link function calls to create direct references to the compiled function in the compiled code, instead of to the function symbol. This feature improves the performance of the compiled code and protects the code against function redefinition at runtime.
- Drop function names to make the compiled code more secure and to decrease program size and load time.
- Drop the names of all local variables and directly link their references. This also makes the compiled code more secure and decreases program size and load time.

The VLISP project management feature allows you to tailor the optimization options to the specific needs of your application. See “Choosing a Compilation Mode” on page 155 to learn more about choosing optimization options.

## Identifying the Input File

If your source file is in the AutoCAD Support File Search Path, you do not have to include the path name when specifying the file name. The search path is set by choosing Tools ► Options from the AutoCAD menu, then clicking the Files tab and selecting Support File Search Path:



For example, if you are compiling the *yinyang.lsp* program file that is in the AutoCAD *Sample\VisualLISP* directory, and Support File Search Path is set as indicated in the previous figure, you can issue the following command to compile the program:

```
(vlisp-compile 'st "yinyang.lsp")
```

If the AutoCAD *Sample\VisualLISP* directory is not in the Support File Search Path, you must include the entire path name when specifying the source file. For example:

```
(vlisp-compile  
'st "c:/program files/autocad T/Sample/VisualLISP/yinyang.lsp")
```

If you omit the file extension from a file name, VLISP assumes the *.lsp* extension.

When specifying the file path name, replace the backslash symbol (\) you normally use for file names with either a forward slash or a double backslash, following the usual AutoCAD convention.

## Naming an Output File

The compiler produces code in the fast-load AutoLISP format (FAS). By default, the output file containing this code has the same name as the input file, but with an extension of *.fas*. You can override the default name by specifying an output file name. For example, to compile *yinyang.lsp* and produce an output file named *GoodKarma.fas*, issue the following command:

```
(vlisp-compile 'st "yinyang.lsp" "GoodKarma.fas")
```

---

**NOTE** If you specify an output file name but do not specify a path name for either the input or the output file, VLISP places the output file in the AutoCAD install directory! This is probably *not* what you want.

---

In most instances, you'll want to specify the full path name of the output file. For example:

```
(vlisp-compile 'st "yinyang.lsp"  
"c:/program files/.../Sample/VisualLISP/GoodKarma")
```

This ensures that the output file is placed in the directory you want it to be.

## Walking through a Compile Example

You can use the *yinyang.lsp* file in the AutoCAD *Sample\VisualLISP* directory to exercise the `vlisp-compile` function.

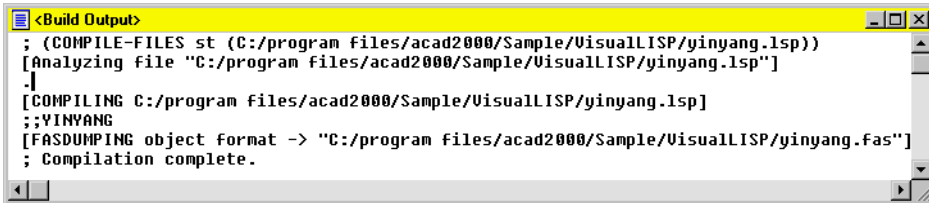
To compile the *yinyang.lsp* sample program

- 1 At the Console prompt, enter the following:

```
(vlisp-compile  
'st "c:/program files/acad2000/Sample/VisualLISP/yinyang.lsp")
```

This command requests a standard mode compile of the *yinyang.lsp* file. No output file name is specified, so the compiled result will be saved in a file named *yinyang.fas* and will be placed in the same directory as the input file (*c:\Program Files\ACAD2000\Sample\VisualLISP*).

- 2 Look at the Build Output window displayed after the command executes. If necessary, scroll up in the window to see all the compiler messages. If the compile completed successfully, the window contains messages like the following:



```
<Build Output>
; (COMPILE-FILES st (C:/program files/acad2000/Sample/VisualLISP/yinyang.lsp))
[Analyzing file "C:/program files/acad2000/Sample/VisualLISP/yinyang.lsp"]
-
[COMPILING C:/program files/acad2000/Sample/VisualLISP/yinyang.lsp]
;;YINYANG
[FASDUMPING object format -> "C:/program files/acad2000/Sample/VisualLISP/yinyang.fas"]
; Compilation complete.
```

During compilation, the compiler prints function names and various messages about each stage of compilation. The first stage is syntax and lexical checking of the source code. If the compiler encounters errors, it issues messages and halts the compilation process. The compiler issues warnings if it encounters expressions it considers dangerous, such as redefining existing AutoLISP functions or assigning new values to protected symbols. If the compiler displays warning or error messages, you can view and edit the source code that caused these messages by double-clicking on the message in the Build Output window.

If compilation is successful, as in the example above, the Build Output window displays the name of the compiled output file.

## Loading and Running Compiled Programs

Compiled AutoLISP programs can be loaded and run from either the VLISP Console window or the AutoCAD Command prompt, or by choosing Tools ► Load Application from the AutoCAD menu. This is true of both *.fas* files and *.vlx* files, which may contain multiple compiled programs. (See “Making Application Modules” on page 117 for information on creating *.vlx* files.)

To run a compiled program from the Visual LISP Console window

- 1 Load the program by invoking the `load` function from the Console prompt. For example, to load the compiled *yinyang* program created in “Walking through a Compile Example” on page 113, enter the following command:  
(load "c:/program files/ACAD2000/sample/VisualLISP/yinyang.fas")

If you do not specify a file extension, `load` first looks for a file with the name you specified (for example, *yinyang*), and an extension of *.vlx*. So if you have previously built an application named *yinyang.vlx*, `load` uses that file instead

of the *.fas* file you just compiled. If no *.vlx* file is found, `load` searches next for a *.fas* file, and finally, if no *.fas* file is found, `load` searches for a *.lsp* file. If you prefer less typing and more clicking, choose **File ► Load File** from the VLISP menu, and use the Load Lisp File dialog box to select the file you want to load. Remember to use the Files of Type pull-down list in this dialog box to specify the type of file you want to load, otherwise VLISP lists only *.lsp* files in the dialog box. You can select from the following types:

- Lisp Source Files (*.lsp* files)
- Compiled AutoLISP Files (*.fas* files)
- VL Packed Application (*.vlx* files)
- All files (lists all files in the specified directory)

Press **Open** to load the selected files.

- 2 At the VLISP Console prompt, enter the name of the function you want to run, enclosing the name in parentheses. For example:

(yi nyang)

VLISP transfers control to AutoCAD to display program prompts and accept user input.

Once you load a program, you can run it from either the AutoCAD Command prompt or the VLISP Console window prompt. Note that if the name of the function you are running begins with `c:`, you can invoke it from the AutoCAD Command prompt as if it were an AutoCAD command, that is, without enclosing the name in parentheses. See “C:XXX Functions” on page 247 for more information on this feature.

Refer to the *AutoLISP Reference* for more information on the `load` function.

## Loading Extended AutoLISP Functions

VLISP provides some extensions to the AutoLISP language that are not loaded automatically when you start AutoCAD. These functions have names that begin with *vla-*, *vlax-*, and *vlr-*. The *vla-* functions implement AutoLISP ActiveX support. The *vlax-* functions provide ActiveX utility and data conversion functions, dictionary handling functions, and curve measurement functions. The *vlr-* functions provide support for AutoCAD reactors. Before you can use any of these functions, you must load the AutoLISP extensions with the following function call:

(vl-load-com)

This function first checks if the AutoLISP extensions are already loaded; if so, the function does nothing, otherwise it loads the extensions.

AutoLISP code that includes calls to *vla-*, *vlax-*, or *vlr-* functions should always begin with a call to `vl-load-com` to ensure that the code will run; it should not be left up to the user to load the extensions. If your application does not call `vl-load-com`, the application may fail.

## Linking Function Calls

The process of compiling function calls results in VLISP creating a loadable module containing in-line copies of some AutoLISP built-in functions. In-line compilation of built-in functions increases the efficiency of the resulting code, but may change the behavior of some tricky AutoLISP programs.

For example, if your program contains an in-line copy of a built-in function, and that function is subsequently redefined, your program does not use the new function definition. A copy of the old definition is part of the program's load module, and that version is called directly. You must recompile your program to pick up the new function definition.

If you include both direct and indirect calls to the same function, your program could end up using different versions for different function calls. This is one reason why combining direct and indirect calls within a single program is not recommended.

When using multiple-file applications in conjunction with direct linking, it is highly recommended that you use the VLISP built-in project management system along with its functions to optimize code automatically. The project management system provides a greater degree of control over compilation and linking of program files than does the `vlisp-compile` function. See “Managing Multiple LISP Files” on page 138 for details.



# Making Application Modules

VLISP provides you with the ability to create a single, standalone executable module for your application. This module incorporates all your application's compiled files, and can include DCL, DVB, and other files that your application may need. Executable VLISP modules are known as VLX files, and are stored in files named with a *.vlx* extension.

A Make Application wizard guides you through the application building process in VLISP. The result of this process is a Make file, which is often referred to by its file extension, *.prv*. The Make file contains all the instructions VLISP needs to build the application executable.

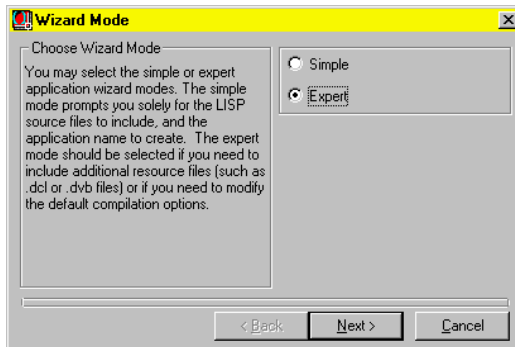
## Creating a New Application

It is recommended you build your application only after you have fully debugged it. Compiler errors during the Make process may prevent the application wizard from completing successfully.

There are two modes of the Make Application wizard: simple and expert. In the simple mode, you need only identify the files in your application and name the files you want to create. The expert mode allows for many additional options.

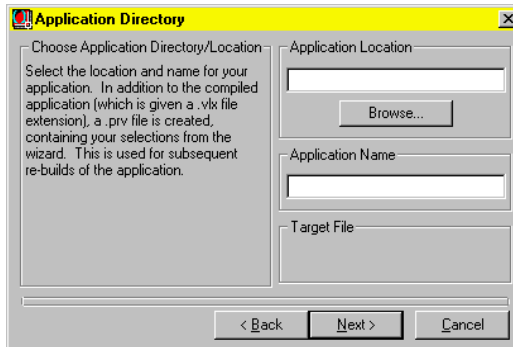
To build an executable file with the Make Application wizard

- 1 Choose File ► Make Application ► New Application Wizard from the VLISP menu to start the Make Application wizard. VLISP displays a Wizard Mode dialog box asking you to choose the mode you want:



Select Expert mode, so you can see all the possible Make options; then press the Next button.

- 2 VLISP displays the following Application Directory dialog box, where you name your application and specify where you want the application files built by Make Application to reside:



The Application Directory dialog box appears in both the Simple and Expert Wizard modes.

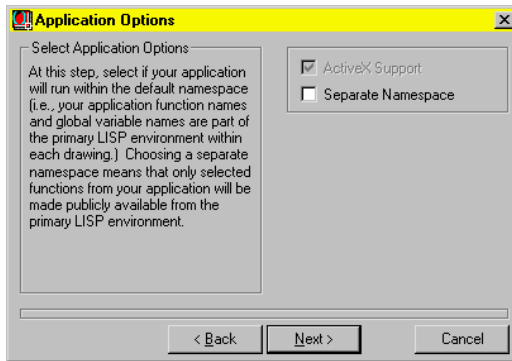
You can enter the full path name in the Application Location field, or press the Browse button and identify the output directory (folder) using a standard Windows dialog box.

The Make Application wizard uses the Application Name when it creates the application executable (.vlx) file and the Make (.prv) file. For example, if you specify an application name of *myapp*, the Make Application wizard creates files *myapp.vlx* and *myapp.prv*.

If you need to go back to a previous Make Application wizard step and change something, press the Back button. Otherwise, press Next to continue.

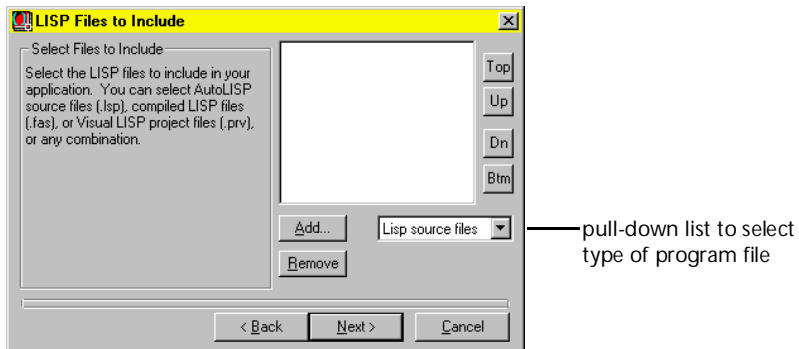
- 3 The Application Options dialog box is displayed when you run the Make Application wizard in Expert mode. In this dialog box, you choose whether you want your application to run in its own namespace or in the namespace of the document from which the VLX is loaded. See “Designing for a Multiple Document Environment” on page 126 for a discussion of namespaces and separate-namespace VLX behavior.

The ActiveX Support option is available if you choose to run your application in a separate namespace. Selecting this option results in automatic loading of AutoLISP ActiveX support functions when the VLX is loaded.



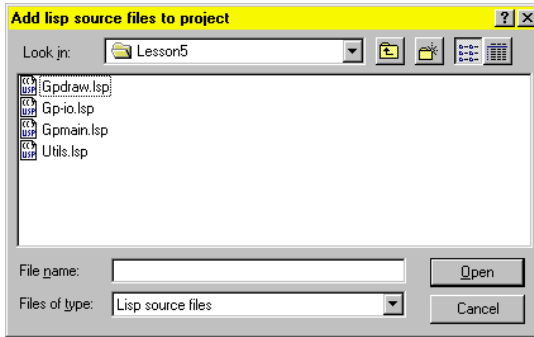
Press Next to continue building the application.

- 4 VLISP displays a dialog box in which you specify the LISP files to be loaded when your application loads.



The LISP Files to Include dialog box appears in both the Simple and Expert Wizard modes.

You can specify AutoLISP source code files, compiled AutoLISP (FAS) files, or a VLISP project file. Click the pull-down button to choose the type of file you want to include, then press the Add button to display the following dialog box for selecting the files:



You can select multiple files using the standard Windows file selection methods. After selecting file names, press Open to add the files to your application. To add more files of a different type, choose the file type from the pull-down list and press Add again.

If you specify AutoLISP source files, VLISP compiles those program files when it builds the application. If you specify a project file, all the project's files are compiled and included in the output module. See "Managing Multiple LISP Files" on page 138 for information on creating and using project files.

To remove files from the application, select the files you no longer want and press the Remove button. You can also select one or more files, right-click, and choose Remove from the shortcut menu.

VLISP loads the application's files in the order they are listed in the List Files to Include dialog box. You may need to reorder the file list. For example, if you call a function at load time, the function must be defined before it is used. In this case, you want to place the file defining that function first. The List Files to Include dialog box contains buttons you can use to move files around in the list. Select a file name, then choose from among the following buttons:

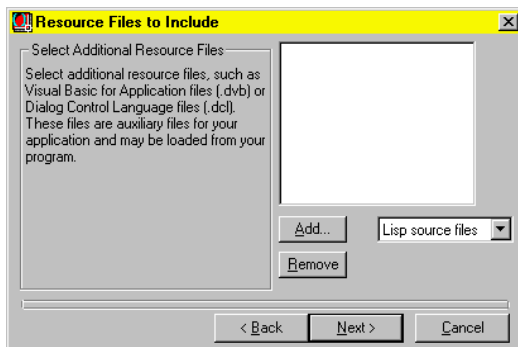
- **Top** Move to the top of the list.
- **Up** Move ahead of the file just above in the list.
- **Dn** Move behind the file just below in the list.
- **Btm** Move to the bottom of the list.

You can also right-click and choose these actions from a shortcut menu.

Note that the load order of project files is specified when you define the project. (See “Changing the Order in Which Visual LISP Loads Files” on page 144 of the “Maintaining Visual LISP Applications” chapter.)

When you have finished specifying your application’s AutoLISP files, press Next to continue to the next step in the Make Application wizard.

- 5 The Resource Files to Include dialog box is displayed when you run the Make Application wizard in Expert mode. If your application includes additional files, such as dialog control language (DCL) files, you can include them in your application’s VLX module by selecting them in the dialog box.



You can specify the following types of resource files:

- AutoLISP source files
- Compiled LISP files
- Visual LISP project files
- DCL files
- DVB files
- Text files

All program files can be loaded by the VLX. If you choose a Visual LISP project file, all files defined in the project files are compiled and included in the VLX.

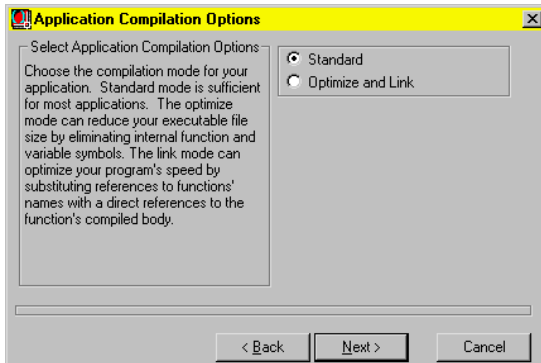
Click the pull-down button to choose the type of files you want to include, then press the Add button to display the dialog box for selecting the files. In the file selection dialog box, you can select multiple files using the standard Windows file selection methods. After selecting file names, press Open to add the files to your application.

To add more files of a different type, choose the file type from the pull-down list and press Add again.

To remove resource files from your application, select the files you no longer want and press the Remove button. You can also select one or more files, right-click, and choose Remove from the shortcut menu.

After selecting resource files for your application, press Next to continue the Make Application process.

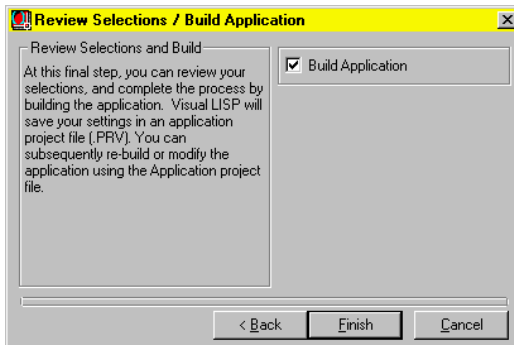
- 6 The Application Compilation Options dialog box is displayed only in Expert mode. You can select the compilation and linkage options for your application in the dialog box.



Refer to “Choosing a Compiler Mode” on page 111 for information on these options.

After selecting your compilation options, press Next to continue to the final step of the Make Application process.

- 7 For the final step in the Make Application wizard, you can tell VLISP to build your application. The Review Selections/Build Application dialog box appears in both Simple and Expert modes.



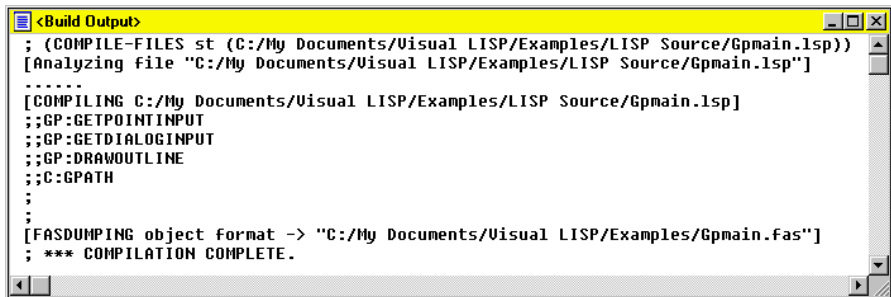
VLISP saves all your application options in a Make (.prv) file. The Make file also includes all the instructions that VLISP needs to build the application. If

you do not elect to build the application now, VLISP can use the Make file to build the application later.

Press Finish to conclude the Make Application process.

## Understanding the Output from Make Application

VLISP executes instructions in a Make file to build an application. Output messages from this process appear in two VLISP windows: the Build Output window and the Console window. The Build Output window contains messages relating to any compilation of AutoLISP source code into *.fas* files. In a successful compile, the output looks like the following:



```
<Build Output>
; (COMPILE-FILES st (C:/My Documents/Visual LISP/Examples/LISP Source/Gpmain.lsp))
[Analyzing file "C:/My Documents/Visual LISP/Examples/LISP Source/Gpmain.lsp"]
.....
[COMPILING C:/My Documents/Visual LISP/Examples/LISP Source/Gpmain.lsp]
;;GP:GETPOINTINPUT
;;GP:GETDIALOGINPUT
;;GP:DRAWOUTLINE
;;C:GPATH
;
;
;
[FASDUMPING object format -> "C:/My Documents/Visual LISP/Examples/Gpmain.fas"]
; *** COMPILATION COMPLETE.
```

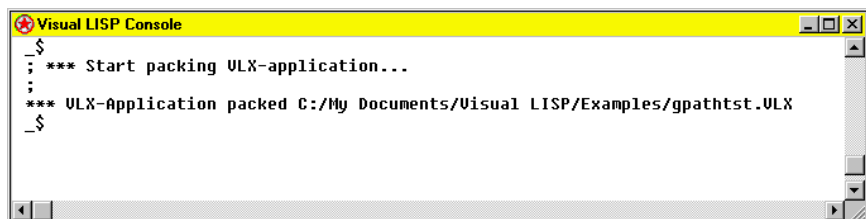
The compiler messages identify the following items:

- The name and directory path of the source files being compiled.
- The functions defined in the source file.

In the above example, four functions are identified: GP:GETPOINTINPUT, GP:GETDIALOGINPUT, GP:DRAWOUTLINE, and C:GPATH.

- The name and path of the output *.fas* files.

The VLISP Console window displays messages relating to the creation of the application executable, the *.vlx* file. If the Make Application process succeeds, the Console window displays the path and file name of the *.vlx*, as in the following example:



```
Visual LISP Console
_$
-;
; *** Start packing VLX-application...
;
; *** VLX-Application packed C:/My Documents/Visual LISP/Examples/gpathtst.VLX
_$
```

## Loading and Running Visual LISP Applications

To execute the functions in a VLX application, you must first load the VLX file using any of the following methods:

- Call the AutoLISP Load function.
- Choose File ► Load File from the VLISP menu.
- Choose Tools ► Load Application from the AutoCAD menu.

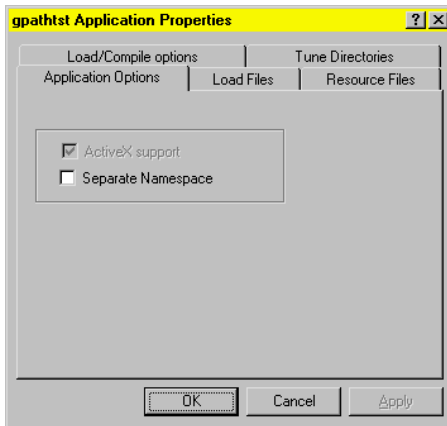
See “Loading and Running Compiled Programs” on page 114 for specific instructions on loading and running application functions.

## Changing Application Options

VLISP allows you to change the way your application is designed. For example, you can change compilation options, or add or remove AutoLISP files from the application.

To change an application’s definition

- 1 Choose File ► Make Application ► Existing Application Properties. VLISP displays a dialog box for you to specify your application’s Make (.prv) file, which is where VLISP stores the application’s properties.
- 2 Specify the name of your application’s Make file, then press Open. VLISP displays the Application Properties dialog box.



- 3 Click the tab of the property you want to change. The tab options are:  
Load/Compile Options      Compile AutoLISP source files using the Standard compile option, or Optimize and Link the files.



- |                     |  |
|---------------------|--|
| Tune Directories    | Identify LISP Object directory and Target directory. The Object directory is where VLISP places <i>.fas</i> and temporary files created by the compiler. Target directory is another name for “Application directory,” which is where Make Application stores the VLX file. If a field identifying a directory is blank, VLISP uses the <i>.prv</i> directory. |
| Application Options | Create a separate-namespace VLX and include ActiveX support. If ActiveX Support is selected along with Separate Namespace, loading the VLX will automatically result in the loading of AutoLISP ActiveX support functions.   |
| Load Files          | AutoLISP source files included in the application.   |
| Resource Files      | Additional resource files included in the application.   |
- 4 After changing a property, press Apply to save the change, or press OK to save the change and exit the Application Properties dialog box.

## Rebuilding an Application

After changing application options or modifying source code, you need to rebuild your application for the changes to take effect.

### To rebuild an application

- 1 Choose File ► Make Application ► Rebuild Application from the VLISP menu.
- 2 Specify the location of your application’s Make file.
- 3 Press Open to rebuild the application.

In rebuilding the application, VLISP recompiles all *.lsp* source files, applying the specified compilation options, and packages your application files into a new *.vlx* file. If your application contains many AutoLISP files, and you have only changed the source code in one or two files, the Make Application option can rebuild your application more efficiently. See the following section for information on using this option.

## Updating an Application

If you change just a small piece of your application’s AutoLISP source code, you can have VLISP rebuild your application VLX while compiling only those files you’ve updated. To effect this type of update, choose File ► Make Application ► Make Application from the VLISP menu, and select your application’s Make file. VLISP rebuilds the application based on

the information contained in the Make file, and automatically compiles any application source files for which either of the following are true:

- There is no compiled (*.fas*) version of the file.
- There is a compiled version of the file, but the source file has been modified since that compile (that is, the date of the source file is more current than the date of the *.fas* file).

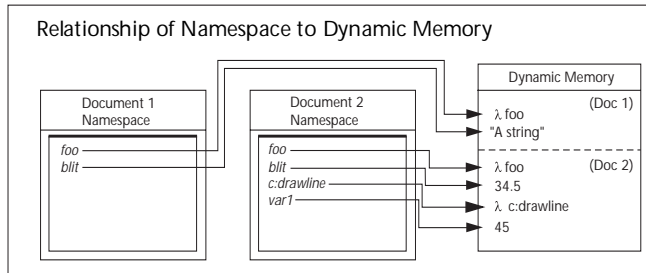
Note that if you change application options (for example, from Standard compile mode to Optimize and Link), you must use the Rebuild Application menu option to create a new VLX with the changes you specified. The Make Application command only checks for changes to AutoLISP source code files, not to application options.

## Designing for a Multiple Document Environment

AutoCAD 2000 introduced the ability to open many drawing documents in a single AutoCAD session. This feature, known as multiple document interface (MDI), provides many benefits to AutoCAD users, such as allowing you to copy objects between drawings and to display several drawings side by side in a single AutoCAD session. When you design an AutoCAD application, you need to understand how open drawing documents relate to one another.

### Understanding Namespaces

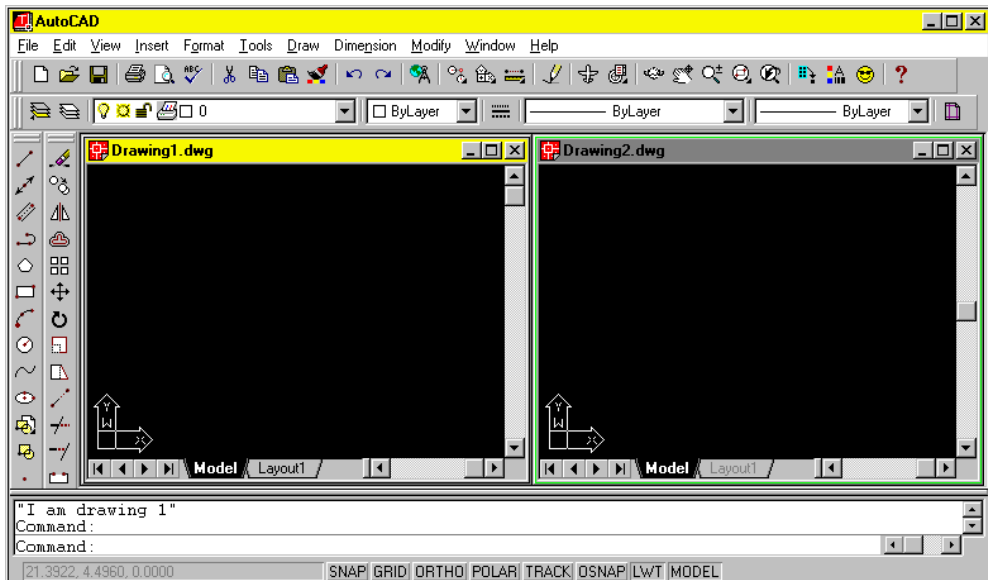
To prevent applications running in one drawing window from unintentionally affecting applications running in other windows, the concept of namespaces was introduced in AutoCAD 2000. A namespace is a LISP environment containing a set of symbols (for example, variables and functions). Each open AutoCAD drawing document has its own namespace. Variables and functions defined in one document namespace are isolated from variables and functions defined in other namespaces.



You can see how this works by trying a simple example.

To see the effect of multiple namespaces

- 1 Open two new drawings in AutoCAD.
- 2 Choose Window ► Tile Vertically from the AutoCAD menu. You should see two open document windows side by side within the main AutoCAD window:



The document's title bar indicates which window is currently active. In the preceding example, *Drawing1.dwg* is the current document.

- 3 Enter the following at the Command prompt:

```
(setq draw1foo "I am drawing 1")
```

This sets the `draw1foo` variable to a string.

- 4 Activate *Drawing2.dwg* (click in the window's title bar).

- 5 See if `draw1foo` contains the value you just set for it:

```
Command: !draw1foo  
nil
```

The variable is `nil` because it has not been set in this document's namespace; you set it in the namespace belonging to *Drawing1.dwg*.

- 6 Enter the following at the Command prompt:

```
(setq draw2foo "I too am a drawing, but number 2")
```

This sets the `draw2foo` variable to a string.

- 7 Activate *Drawing1.dwg*.

- 8 Test the values of variables `draw1foo` and `draw2foo`:

```
Command: !draw1foo  
"I am drawing 1"  
Command: !draw2foo  
nil
```

The `draw1foo` variable contains the value you set for it, but `draw2foo` is `nil` because you did not set it to a value in the current namespace; you set a different variable of the same name in *Drawing2.dwg*'s namespace.

VLISP provides ways for you to share variables between namespaces, but you must take explicit action to do so. (See "Sharing Data between Namespaces" on page 133.)

Like variables, functions defined in an AutoLISP file are known only to the document that was active when the file was loaded. The functions in the file are loaded in the current document's namespace and are known only to that document.

To see how functions are affected by multiple namespaces

- 1 Load a LISP file from either the AutoCAD Command prompt or the VLISP Console prompt. For example:

```
(load "yinyang.lsp")
```

- 2 Invoke the function.
- 3 Open a second drawing window.
- 4 With the second drawing window active, try invoking the function again. The response will be an error message saying the function is not defined.

You can use the `vl-load-all` function to load the contents of an AutoLISP file into all AutoCAD drawing documents. For example, the following command causes the contents of the *yinyang.lsp* file to be loaded into all open documents, and into any documents opened later in the AutoCAD session:

```
(vl-load-all "yinyang.lsp")
```

The `vl-load-all` function is useful for testing new functions in multiple documents, but in general you should use *acaddoc.lsp* to load files that are needed in every AutoCAD document.

## Running an Application in Its Own Namespace

You can define a namespace for a VLX application in VLISP. A VLX application defined in this manner is referred to as a separate-namespace VLX. When you load a separate-namespace VLX, it runs in its own namespace, not the namespace of the document from where you loaded the VLX. The option to define a VLX application with its own namespace is part of the Make Application procedure (see “Making Application Modules” on page 117).

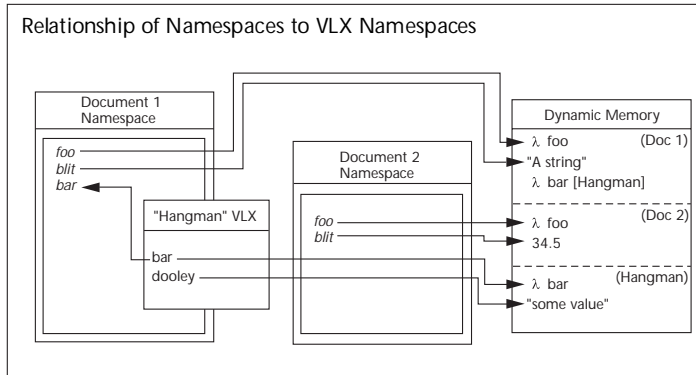
If you try to load a separate-namespace VLX that is already loaded, you’ll receive an error indicating this. Use the `vl-unload-vlx` function to unload the application. The function’s syntax is

```
(vl-unload-vlx "appname")
```

For *appname*, specify the file name of the VLX, without the path or the *.vlx* extension.

Variables and functions defined in a VLX application’s namespace are known only to the application, not to the drawing document that was active when the application was loaded. This allows you to protect your variables from accidentally—or intentionally—being overwritten by other applications or users.

A VLX application can export function names to a document namespace to enable those functions to be accessed within the context of that document. The following diagram illustrates how this works:



The diagram shows an AutoCAD session containing two open drawing documents. A VLX application named “hangman” is loaded with respect to *Document1* (for example, a user opened *Document1* and then loaded the VLX application from the AutoCAD Command prompt). The hangman application established its own namespace and declared the bar function and the dooley variable in that namespace. The VLX exported the bar function to *Document1*’s namespace. When a user invokes bar from *Document1*, bar runs in the application’s namespace. The bar function is unknown to *Document2*, and neither document has access to the dooley variable (because the VLX did not export it). You can load another instance of the hangman VLX into *Document2*, but this instance will have its own namespace and its own copies of bar and dooley.

---

**NOTE** When you load a VLX file that has *not* been defined as having its own namespace, the environment is similar to that of a loaded file. All functions and variables defined in the VLX are loaded in the document’s namespace.

---

## Making Functions Available to Documents

By default, functions defined in a separate-namespace VLX are not exposed to the document namespace from which the VLX is loaded. You must use the `vl-doc-export` function to expose functions to document namespaces. When issued from a VLX that runs in its own namespace, `vl-doc-export` exposes the specified function to any document namespace that loads the VLX. The `vl-doc-export` function accepts a single argument, a symbol identifying the function name. For example, look at the following code:

```
(vl-doc-export 'kertrats)
(defun kertrats ()
  (princ "This function goes nowhere")
)
```

This example defines the `kertrats` function, which simply prints a message. The `defun` for the function is preceded by a `vl-doc-export` call that causes the function to be exported to the document namespace.

### To see how `vl-doc-export` works in a separate-namespace VLX

- 1 In a VLISP text editor window, copy the following code into a file:

```
(defun kertrats ()
  (princ "This function goes nowhere")
)
```

Note that this code does not contain a call to `vl-doc-export`.

- 2 Save the file you just created.
- 3 Use the VLISP Make Application wizard to build a VLX from your program file. Specify the following wizard options:
  - Wizard mode: Expert
  - Application name: `doctest`
  - Application options: Separate-namespace
  - Compilation options: Optimize
- 4 From either the AutoCAD Command prompt or the VLISP Console window prompt, load the *doctest* VLX file.
- 5 Try running the `kertrats` function.
 

You should receive an error message indicating the function is not defined.
- 6 Add the following line of code to the beginning of your program file:
 

```
(vl-doc-export 'kertrats)
```
- 7 Save the file, then rebuild the application.
- 8 Use `vl-unload-vlx` to unload the VLX, then load and run the VLX again. This time, `kertrats` should run successfully.

You can issue a `vl-doc-export` call outside the context of a separate-namespace VLX application, but it has no effect.

The `vl-list-loaded-vlx` function returns a list of all separate-namespace applications associated with the current document. For example:

```
_$ (vl-list-loaded-vlx)
(DOCTEST)
```

To determine what functions have been exported from a separate-namespace application into the current document, use `vl-list-exported-functions`.

When calling this function, you must pass it a string naming the application you are checking. For example, the following command returns a list of the functions exported by the doctest application:

```
_$(vl -list-exported-functions "doctest")  
("KERTRATS")
```

The results show that a single function, `kertrats`, was exported from `doctest` to the current document's namespace.

---

**NOTE** Currently, if separate namespace VLX A associated with document A loads separate namespace VLX B, then all of VLX B's exported functions are automatically defined in document A. This behavior will not be supported in future releases of AutoCAD. Note also that VLX B's exported functions are not defined in VLX A until VLX A issues an explicit import (see the next topic).

---

## Making Separate-Namespace Functions Available to Other VLX Applications

Functions defined in one separate-namespace VLX are not exposed to any other separate-namespace VLX applications. If a function has been exported through `vl -doc-export`, you can use the `vl -doc-import` function to make the function available to another separate-namespace VLX.

## Referencing Variables in Document Namespaces

Variables defined in a separate-namespace VLX are not known to the document namespace associated with the VLX. However, a separate-namespace VLX can access variables defined in a document namespace using the `vl -doc-ref` and `vl -doc-set` functions.

The `vl -doc-ref` function copies the value of a variable from a document namespace. The function requires a single argument, a symbol identifying the variable to be copied. For example, the following function call copies the value of a variable named `aruhu`:

```
(vl -doc-ref ' aruhu)
```

If executed within a document namespace, `vl -doc-ref` is equivalent to the `eval` function.

The `vl -doc-set` function sets the value of a variable in a document namespace. The function requires two arguments: a symbol identifying the variable to be set, and the value to set for the variable. For example, the following function call sets the value of a variable named `ul us`:

```
(vl -doc-set ' ul us "Go boldly to noone")
```



If executed within a document namespace, `vl-doc-set` is equivalent to the `setq` function.

To set the value of a variable in all open document namespaces, use the `vl-propagate` function. For example, the following function calls `set` a variable named `fooyal` in all open document namespaces:

```
(setq fooyal "Go boldly and carry a soft stick")  
(vl-propagate 'fooyal)
```

This command not only copies the value of `fooyal` into all currently open document namespaces, but also causes `fooyal` to automatically be copied to the namespace of any new drawings opened during the current AutoCAD session.

## Sharing Data between Namespaces

VLISP provides a blackboard namespace for communicating the values of variables between namespaces. The blackboard is a namespace that is not attached to any document or VLX application. You can set and reference variables in the blackboard from any document or VLX. Use the `vl-bb-set` function to set a variable, and use `vl-bb-ref` to retrieve a variable's value.

For example, the following command sets the `foobar` blackboard variable to a string:

```
Command: (vl-bb-set 'foobar "Root toot toot")  
"Root toot toot"
```

The `vl-bb-ref` function returns the specified string. The following example uses `vl-bb-ref` to retrieve the value of `foobar` from the blackboard:

```
Command: (vl-bb-ref 'foobar)  
"Root toot toot"
```

Note that these functions require you to pass a symbol naming the variable you are referencing (*'var-name*), not the variable name (*var-name*).

Setting or retrieving variable values in the blackboard namespace has no effect on variables of the same name in any other namespace.

To demonstrate that document variables are unaffected by blackboard variables

- 1 From the VLISP Console window (or the AutoCAD Command prompt), use `vl -bb-set` to set the `*example*` blackboard variable.

```
_ $ (vl -bb-set ' *example* 0)  
0
```

The `*example*` variable is set to 0 in the blackboard namespace.

- 2 Use `vl -bb-ref` to verify the value of the variable you set in the previous step.

```
_ $ (vl -bb-ref ' *example*)  
0
```

- 3 See what value `*example*` has in the current AutoCAD document.

```
_ $ *example*  
nil
```

The `*example*` variable is `nil` because it has not been set in the document namespace.

- 4 Set `*example*` in the current document.

```
_ $ (setq *example* -1)  
-1
```

The `*example*` variable is set to -1 in the document namespace.

- 5 Check the current value of `*example*` in the blackboard.

```
_ $ (vl -bb-ref ' *example*)  
0
```

The blackboard variable named `*example*` is still set to the value assigned in step 1; setting the document variable of the same name in step 4 had no effect on the blackboard.

VLISP also provides the `vl -doc-set` and `vl -doc-ref` functions to set and retrieve document namespace variables from a separate-namespace VLX, and `vl -propagate` to set the value of a variable in all open document namespaces. These functions are described in “Referencing Variables in Document Namespaces” on page 132.

## Handling Errors in an MDI Environment

By default, each document namespace is provided with its own `*error*` function, which is defined as follows:

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

A VLX application running within a document namespace shares the default error-handler function. You may want to add error-handling logic to your application.

## Handling Errors in a VLX Application Running in Its Own Namespace

For VLX applications executing within their own namespace, you can either use the default error function or you can define an error handler specifically for the application.

If you define an error handler for a VLX running in its own namespace, you can call the `vl-exit-with-error` function to pass control from the VLX error handler to the document namespace's `*error*` function. The following example uses `vl-exit-with-error` to pass a string to the document's `*error*` function:

```
(defun *error* (msg)
  ... ; processing in VLX namespace/execution context
  (vl-exit-with-error (strcat "My application bombed! " msg)))
```

A VLX `*error*` handler can use the `vl-exit-with-value` function to return a value to the document namespace from which the VLX was invoked. The following example uses `vl-exit-with-value` to return the integer value 3 to the program that called the VLX from the document namespace:

```
(defun *error* (msg)
  ... ; processing in VLX-T namespace/execution context
  (vl-exit-with-value 3))

(vl-doc-export 'foo)
(defun foo (x)
  (bar x)
  (print 3))
(defun bar (x) (list (/ 2 x) x))
```

Any instructions pending at the time the error occurred are flushed.

If your VLX namespace error handler does not use either `vl-exit-with-error` or `vl-exit-with-value`, then control returns to the command prompt after execution of the error handler. You can only call `vl-exit-with-error` and `vl-exit-with-value` in the context of a VLX application's error handler; it is an error to invoke these functions in any other situation.

## Limitations on Using AutoLISP in an MDI Environment

When using native AutoLISP in an MDI environment, you can only work with one drawing document at a time. Although AutoLISP provides support for exchanging variables and exposing functions to multiple namespaces, you cannot, for example, run a function in one document namespace and issue `entmake` to create an entity in another document namespace. AutoLISP does not support accessing information across multiple drawings.

You can access multiple document namespaces using ActiveX automation, and AutoLISP provides access to ActiveX methods (see chapter 6, “Working with ActiveX”). However, accessing multiple documents with ActiveX is an unsupported feature of AutoLISP. For example, an AutoLISP program running in the context of document *A* can change the active document to document *B* by calling `vlax-put-activatedocument`. Changing the active document, though, immediately suspends execution of the program. The program may resume execution if the user activates the window containing document *A* but the system will be in an unstable state and likely to crash.

---

**WARNING!** If you do use ActiveX to work in MDI, be aware that if you close all AutoCAD drawings you lose access to AutoLISP and will cause an exception.

---

# Maintaining Visual LISP Applications

This chapter describes how you can maintain large applications containing multiple files by defining the application as a VLISP project. Aside from defining the components in your application, you can use VLISP projects to define compiler options for the application. This chapter describes the various compiler options and the consequences of each, and tells you how to override these options for individual files in a project.



## In This Chapter

# 5

- Managing Multiple LISP Files
- Defining a Project
- Working with Existing Projects
- Optimizing Application Code

# Managing Multiple LISP Files

Many program examples you have seen in this document have been small, standalone AutoLISP files. Typical AutoLISP applications, however, consist of larger files with many lines of code. An application may include many source code files. After compiling the programs in such an application, you also have a number of FAS files to track.

As the number of application files grows, it becomes more difficult to maintain an application. Determining when you need to recompile files after source code changes can be a challenge. VLISP provides functions that greatly simplify the process of managing multiple-file applications.

## Understanding Visual LISP Projects

To aid you in the process of maintaining multiple-file applications, VLISP provides a construct called a project. A VLISP project contains a list of AutoLISP source files, and a set of rules on how to compile the files. Using the project definition, VLISP can do the following:

- Check which *.lsp* files in your application have changed, and automatically recompile only the modified files. This procedure is known as a Make procedure.
- Simplify access to source files by listing all source files associated with a project, making them accessible with a single-click.
- Help you find code fragments by searching for strings when you do not know which source files contain the text you're looking for. VLISP limits the search to files included in your project.
- Optimize compiled code by directly linking the corresponding parts of multiple source files.

Before discussing how to define and use VLISP projects, it may help to introduce file types used in VLISP.

### LISP, FAS, and Other File Types

The basic file type in VLISP is the AutoLISP source file. Typically, AutoLISP source files are named with an *.lsp* file extension. You have seen *.lsp* files used frequently in previous chapters.

The FAS (*.fas*) file type was introduced in the previous chapter, “Building Applications.” FAS files are compiled AutoLISP files. These files load faster than AutoLISP source files and are more secure because their contents are not intelligible to users.

Here is a brief summary of the types of files used by the VLISP project management feature:

#### Visual LISP project file types

File ext.	Type of file	Function
<i>fas</i>	Compiled AutoLISP code	Compiled AutoLISP programs. May be loaded and run, or compiled into VLX modules.
<i>lsp</i>	AutoLISP source code	Program source files.
<i>ob</i>	Object code	Used internally by VLISP, these files contain compiled AutoLISP code used in building FAS files.
<i>pdb</i>	Project database	Used internally by VLISP, these files contain symbol information used by the compiler.
<i>prj</i>	Project definition	Contains the location and names of all source files that build the project, as well as certain parameters and rules on how to create the final FAS files.

In addition to the files recognized by the project manager, VLISP either creates, processes, or recognizes a number of additional types of files, as summarized below:

#### Additional Visual LISP file types

File ext.	Type of file	Function
<i>dsk</i>	Desktop save	Contains VLISP environment and window settings. (Note: Do not edit this file unless you are certain you know what you are doing.)
<i>_xx</i>	Backup files	Backup copies of edited files, maintained by the VLISP editor. Backup files contain the same name as the original, except that the file extension begins with the underline character ( _ ) and is followed by the first two characters of the original file's extension. For example, the backup file of an .LSP file has an <i>._LS</i> extension; the backup of a DCL file has a <i>._DC</i> extension.
<i>vlx</i>	Standalone applications	Standalone AutoCAD applications, which can be created using the VLISP Make Application wizard.

### Additional Visual LISP file types (*continued*)

File ext.	Type of file	Function
<i>c, cpp, cch, hpp, hh</i>	Language source files	Contain program source code. The VLISP editor recognizes the syntax of these files and color-codes reserved words, strings, and numbers.
<i>dcl</i>	Dialog control language	Contains definitions of AutoCAD dialog boxes. VLISP can preview these files, and you can include them in Visual LISP executable (VLX) files.
<i>prv</i>	Make application	Defines the files and options used to build a VLX application with the VLISP Make Application wizard.
<i>sql</i>	Structured query language	Contains SQL statements. The VLISP text editor recognizes this file type and color-codes the text according to SQL syntax rules.

## Defining a Project

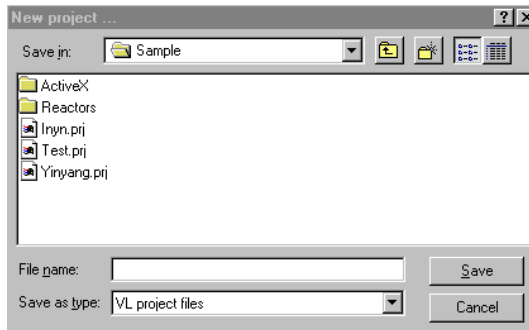
To demonstrate the use of projects in VLISP, you can use the sample programs supplied with the VLISP Tutorial. This code is available on the AutoCAD installation CD, but the tutorial files are only included in your installation if you choose a Full install, or if you choose Custom install and select the Samples item. If you have already installed AutoCAD and did not install the samples, you can rerun the install, choose Custom, and select only the Sample item.

The sample files used in this chapter are in the *Tutorial\VisualLISP\Lesson5* folder of the AutoCAD install directory. The files are:

- *Gpmain.lsp*
- *Gpdraw.lsp*
- *Gp-io.lsp*
- *Utils.lsp*

To create a VLISP project, choose Project ► New Project from the VLISP menu:

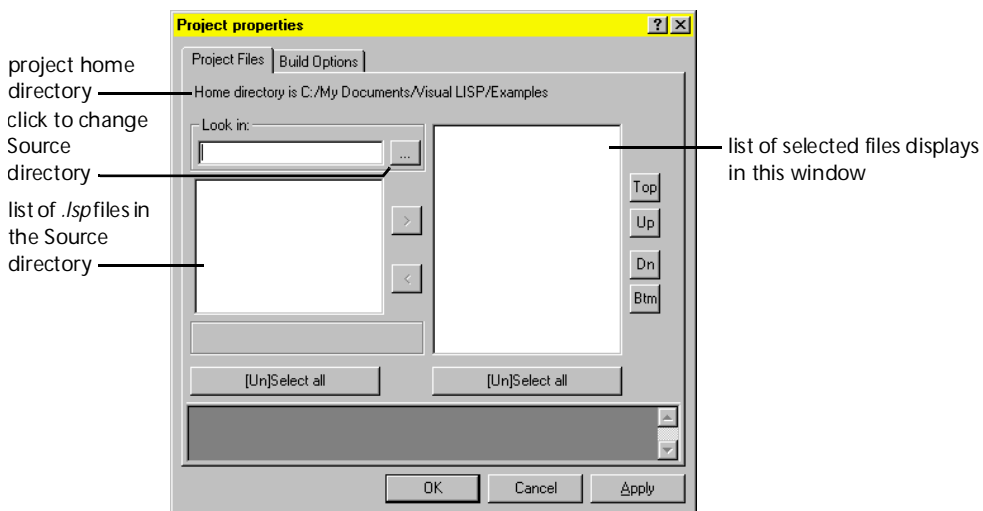




VLISP displays a standard Windows dialog box for you to specify a file path and name. For the example in this chapter, the project name is *Tutorial*. VLISP assigns a *.prj* extension to the project file name.


## Assigning Project Properties

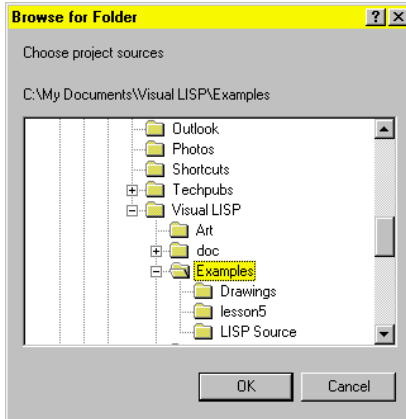
The Project Properties dialog box displays after you specify a project file name:



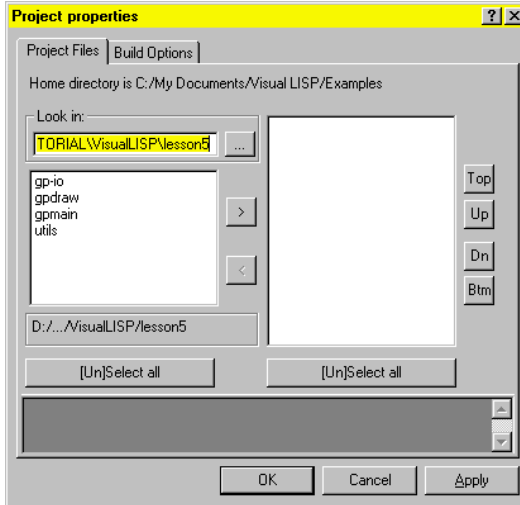
## Selecting the Files to Include in a Project

There are two tabs in the Project Properties dialog box. Choosing the Project Files tab displays the window shown above. In this window, you specify the AutoLISP source files for the project.

The project's home directory is identified just below the tabs. This is where the project file (*tutorial.prj*) resides. In this example, the home directory is *c:\My Documents\VisualLISP\Examples*. That's not the directory containing the tutorial sample files, though. To identify the source directory, press the  button:



Use the Browse for Folder window to identify the location of the project source files. If you select the *Lesson5* directory, the Project Properties dialog box looks like the following:

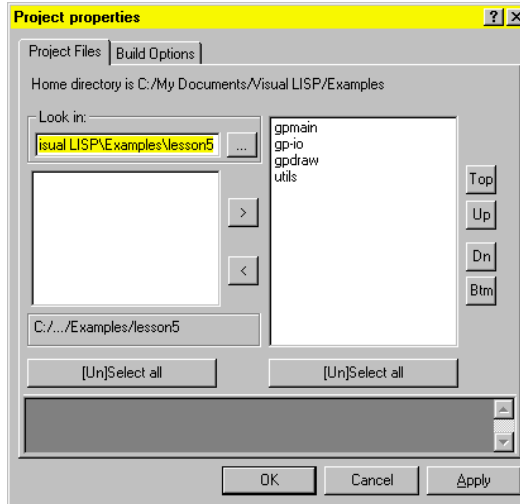


VLISP lists all files in the directory having an *.lsp* extension (but does not display the extension in the list). The window is designed so that, by default, you can select multiple file names by just choosing each name. You do not

have to press and hold CTRL to select more than one file. To clear a selected name, just choose it again.



To include all the listed files in your project, press the button labeled “(Un)Select all,” then choose the right arrow button. VLISP moves the file names to the window on the right:



To remove a file from the project, select the file's name in the right window and click the left arrow button.

## Identifying the Path Name of Project Files

The list of included files does not identify the path name of each file (nor does the Look In field; this just identifies the path of the files listed in the left window). Because you can include files from multiple directories in your project, you need to be able to identify the path name of each file. You can do this by highlighting one or more file names and right-clicking to display a shortcut menu:



To display the full path name and the size (in bytes) of source files in the project, choose Log Filenames and Size from the shortcut menu. The information appears in a small, scrollable window near the bottom of the Project Properties dialog box:



If a file is in the Home directory shown in the Project Properties dialog box, VLISP does not spell out its path name. Use the scroll bar to see information about all the files in the project.

Note that you cannot include two files of the same name in a project, even if they are in different directory paths.

## Changing the Order in Which Visual LISP Loads Files

The shortcut menu for the list of included files also provides commands to move files up and down in the list, and to sort the list by file name or by full path name. VLISP loads the project's files in the order in which they are listed. Sometimes the load order is important. For example, you might have an initialization file that defines global variables needed by all the other program files, and thus must be loaded first. You could select that file name and choose Move to Top to place it first in the project's file list.

You can also use buttons in the Project Properties dialog box to move files around in the list: Top (move to top), Up (move up), Dn (move down), and Btm (move to bottom).

For the tutorial project, the *gpmain.lsp* file should be loaded last. It contains the following instructions at the end of the file:

```
(princ "\nType GPATH to draw a garden path.")
(princ)
```

This results in a prompt telling users how to invoke the application. If VLISP loads *gpmain.lsp* last, these instructions will display at the AutoCAD Command prompt.

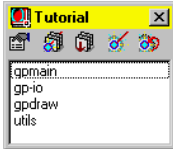
After you move any needed files, press the Apply button.

## Choosing Compiler Build Options

The Build Options tab displays a dialog box in which you can specify compiler options to VLISP. This topic is covered in the “Choosing a Compilation Mode” section later in this chapter. For now, choose OK to close the Project Properties dialog box.

# Using the Project Window to Work with Project Files

When you open a VLISP project, VLISP displays a window listing the files in the project:



By default, VLISP lists the project members in the order in which they will be loaded (as defined in the Project Properties dialog box). You can change this order by choosing Arrange Files from the shortcut menu for this window (see below).

The project name appears in the window title bar. Below the title bar are five icons. Each icon is a button that performs a function. The buttons and their functions are as follows:



Project Properties

Displays the Project Properties dialog box for the project. This allows you to view the full path name of each file in the project, add, remove, and reorder project files, and view and change project compiler options.



Load Project FAS

Loads all compiled (.fas) files for the project.



Load Source Files

Loads all the project source files, making them available to be run.



Build Project FAS

Compiles all project source files that have been modified since their last compile.



Rebuild Project FAS

Recompiles all project source files, whether or not they have changed since their last compile.

If you right-click within the file list of the Project Properties dialog box, VLISP displays a shortcut menu. Many of the functions available from the project shortcut menu can also be accomplished in other ways. For example, you've already seen how to add files to projects and remove files from projects. Choosing Remove File from the shortcut menu is a quick way of removing a file from a project, while choosing Add File merely brings you to the Project Properties dialog box.

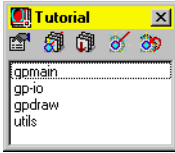
The following summarizes the commands on the shortcut menu:

Edit	Edits the source code of the selected project members.
Add File	Opens the Project Properties dialog box to add files to the project.
Remove File	Removes the selected members from the project.
Load	Loads the FAS file for the selected project members. If no FAS file exists for a member, load the AutoLISP source file.
Load Source	Loads the <i>.lsp</i> file for the selected project members.
Check Syntax	Checks AutoLISP syntax of the source code for the selected members.
Touch	Indicates that the selected source files have been updated, but make no change to the files. This causes VLISP to recompile these programs the next time you ask to compile all changed project files.
Arrange Files	Sorts the project member list, according to one of the available suboptions (load order, name, type, or date).
Multiple Selection	Tells VLISP whether or not to allow selection of multiple members from the list in the Project Properties dialog box. If this option is selected, multiple selection is allowed.
[Un]Select All	Selects all members of the project list, if none are currently selected. If any members are currently selected, this command cancels their selection.
Close Project	Closes the project.
Save Project As	Saves the project.

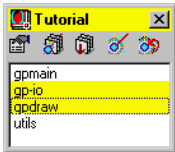
## Selecting Multiple Project Members

The Multiple Selector menu item is available only from the Project Properties dialog box shortcut menu. Choosing this option allows you to select multiple members from the list in the Project window. If the option is selected, a check mark appears next to the Multiple Selector item on the menu. Click on the menu item to toggle it on and off.

If Multiple Selector is in effect, clicking a member name in the Project Properties dialog box acts as a toggle to select or deselect the member. For example, none of the members listed in the following window is selected:



If you click on the name GP-IO, then click on the name GPDRAW, both are selected.

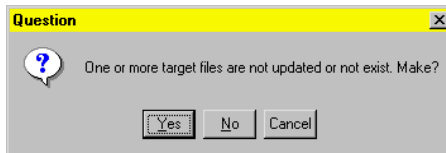


This is unlike the default Windows behavior, where selecting the second list item cancels the first item's selection, unless you press CTRL while selecting the item.

You can also use the Project Properties dialog box shortcut menu to select all members of the project or cancel selection of all members. If no members are currently selected, right-click and choose [Un]Select All to select all the members. If any or all members are already selected, [Un]Select All cancels all selections.

## Loading Project Files

To load the project's compiled program files, click the Load Project FAS button. This allows you to run the application. If VLISP detects that some of the source files do not exist in compiled format, it displays a message and asks if you want to compile those files:



If you choose Yes, VLISP attempts to compile all *.lsp* files that do not have a corresponding *.fas* file. If you choose No, VLISP loads all FAS files it finds for the project, and loads the AutoLISP source for the remaining project files. Press Cancel to abort the load operation.

To load all project source files, instead of their compiled versions, click the Load Source Files button. Remember that debugging breakpoints may be saved within source code files but are removed from the compiled version of

the code. You might want to load source files to debug changes you've made to your programs.

Using the Project Properties dialog box shortcut menu, you can choose to load just selected files. Select the files you want to load, then right-click and choose Load to load the FAS files, or choose Load Source to load the source code. Note that if you choose Load and a FAS file does not exist for a selected file, VLISP loads the AutoLISP source file instead.

---

**NOTE** The *Lesson5* example from the VLISP Tutorial requires a DCL file to run successfully. The DCL file is included in the *Lesson5* folder, but you cannot define a DCL file as part of a VLISP project. To run this example successfully you must copy the DCL file to a directory in the AutoCAD Support File Search Path. You can also define the DCL file as an application component, using the VLISP Make Application wizard. Using this method, the file does not have to be in the AutoCAD search path. "Including a Project in a Visual LISP Application" on page 151 demonstrates how to define an application composed of a VLISP project and supporting files, such as DCL.

---

## Compiling and Recompiling Project Files

One key advantage in defining VLISP projects for your applications is that it provides an efficient method of updating compiled code. You can elect to have VLISP recompile all source files that have changed since the last time they were compiled. By choosing this option, you ensure all FAS files in your application correspond to the latest versions of the program source code. At the same time, you save time by avoiding unnecessary compiles. To invoke this feature, click the Build Project FAS button in the Project Properties dialog box.

You can also choose to recompile all the programs in your project, whether or not they have changed. Click the Rebuild Project FAS button to enable this feature.

## Editing Project Files

To edit the source file of a project member, select the member from the list in the Project window, then right-click and choose Edit. If the Multiple Selector option is on, you can select multiple members, and VLISP will open a text editor window for each.

---

**NOTE** If the Multiple Selector option is not turned on, you can simply double-click a member name to edit it.

---



## Saving and Closing the Project

To save the project properties you defined or modified, right-click in the Project window and choose Save Project As from the shortcut menu. VLISP displays a list of project files. You can either select the name of the current project file to update its contents, or enter a new file name to save the changes as a new project.

When you are finished working with a project, right-click in the Project window and choose Close Project. Note that this only closes the *.prj* file; any project files that are open in VLISP editor windows remain open.

---

**NOTE** If you close the Project Properties dialog box by clicking the Close button, this does not close the project itself. The Project is still open, and you can reopen a Project window for it by choosing it from the Project menu, as described in the next section, “Opening a Project.”

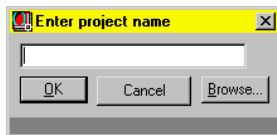
---

## Working with Existing Projects

Some VLISP features described in previous chapters have special application with VLISP projects. The features described in this section are the text editor search functions and the Make Application wizard.

### Opening a Project

To open an existing project, choose Project ► Open Project from the VLISP menu:



If the project file you want to open is in the current directory, you can simply enter the project name here. If the project file is not in the current directory, or if you don't know what the current directory is, press the Browse button to obtain a standard Open dialog box.

Note that you can have more than one project open at a time. You can view a list of all open projects by choosing the Project menu and looking at the bottom of the menu displayed:



At any time, only one of the projects is active. The check mark in front of the project name indicates the active project. The commands in the Project menu, such as Load and Build, apply to the active project. These commands work the same when selected from a Project window, as described earlier in this chapter.

If you attempt to open a project that has the same name as the active project (that is, the project file has the same name, but is in a different directory than the current active project), VLISP displays a message box asking you if you want to “relocate the project definition.” If you choose “Yes,” VLISP loads the new project file and replaces the active project. If you choose “No,” VLISP does not load the new project file, leaving the current active project in place.

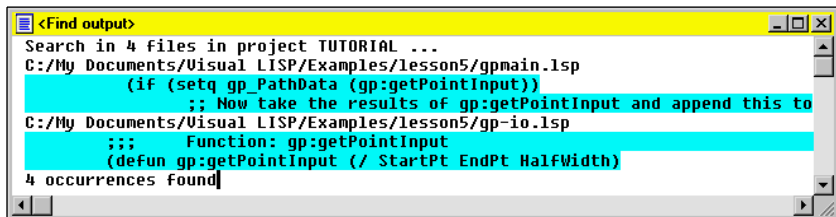
## Finding a String in Project Source Files

The VLISP text search function, described in “Searching for Text” on page 37, provides you with the ability to search all of a project’s source files for a string of text.

For example, suppose in reviewing *gpmain.lsp* you see calls to a function called `gp: getPointInput`, and you cannot remember in which source file this function is defined. To search for it, choose Search ► Find from the VLISP menu. In the Find dialog box, select Project in the list of Search options:



A Project selection field now appears at the bottom of the Find dialog box. If the name of the project you want to search is not already displayed in this field, choose it from the pull-down list. Choose the Find button to perform the search. VLISP displays the results in a Find Output window like the following:



The output shows that four files were searched (there are four source files in the project), and four occurrences of `gp:getPointInput` were found. The occurrences were found in two files; the `defun` for the function is in *gp-io.lsp*. You can open an editor window for the file by double-clicking anywhere within the highlighted text in the Find Output window. You can also press **SHIFT + F11** to display the first source location at which the text string was found, and then repeatedly press **F11** to view subsequent occurrences in the source files.

## Including a Project in a Visual LISP Application

After you've made changes to an application's source files, you'll want to incorporate those changes in the application's executable file. "Making Application Modules" on page 117 showed you how to keep individual application files synchronized, so that the application executable contained the latest versions of all its source files. Defining the application files as members of a single project simplifies this process further. Instead of listing every source file in your application's Make file, you can point to the VLISP project file and use the project file to identify your source files.

### To define an application that includes a Visual LISP project

- 1 Choose File ► Make Application ► New Application Wizard from the VLISP menu to start the Make Application wizard.
- 2 Select Expert mode, then press Next.
- 3 Identify the directory path in which you want VLISP to save your application files, and enter a name for the application.

Press Next to continue.

- 4 Press Next to accept the default application options.
- 5 In the LISP Files to Include dialog box, click the pull-down menu to the right of the Add button and choose Visual LISP project file as the type of file to include.

Press the Add button to display the Add Visual LISP Project Files dialog box.

- 6 Specify the *.prj* file you created for the *Lesson5* tutorial example, then press Open to add the file to your application. All the project's files will be included in the application.

Press Next to continue to the next step in the Make Application wizard.

- 7 In the Resource Files to Include dialog box, click on the pull-down to the right of the Add button and choose DCL files, then press the Add button to display the dialog box for selecting the DCL files.

Select the DCL file in the *Tutorial\VisualLISP\Lesson5* directory, then press Open to add the file to your application.

Press Next to continue the Make Application wizard.

- 8 Accept the default options in the remaining steps and complete the Make Application process.

If you add files to the VLISP project you included in your application, the new files are automatically included the next time you build the application.

# Optimizing Application Code

The optimization features of VLISP can produce more efficient compiled files, and are useful as your programs grow in size and complexity. With the VLISP project management feature, you can tailor the optimization options to the specific needs of your application.

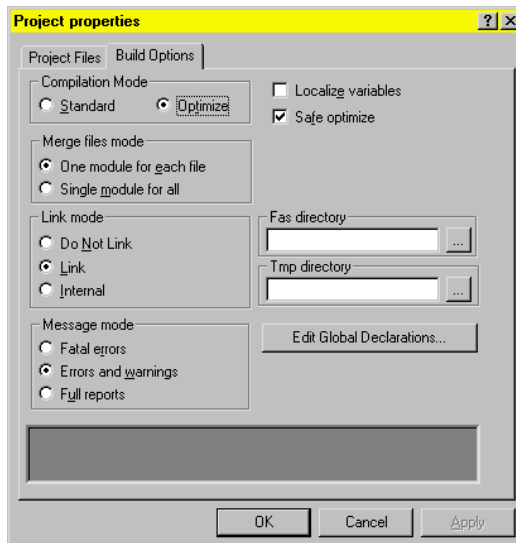
## Defining Build Options

VLISP provides a number of options for compiling and linking a project's source code that affect the efficiency of the compiled code. For example, you can specify whether to create a separate FAS file for each source file or merge all compiled files into a single FAS file. You can choose to have the compiler remove the names of all local symbols from your compiled files. You specify these options in the Project Properties dialog box.

To specify project build options



- 1 Open the project file (choose Project ► Open Project from the VLISP menu).
- 2 Press the Project Properties button in the Project window toolbar.
- 3 Select the Build Options tab in the Project Properties dialog box. VLISP displays the Build Options dialog box:



Some of the build options require extensive background information, which is provided in the following sections of this chapter. The build options are:

Compilation Mode	Choose between standard and optimized compilation. Optimized compilation creates smaller and faster programs but is not suited for every project. See “Choosing a Compilation Mode” on page 155 for more information on this topic.
Merge Files Mode	<p>Tell the compiler whether to create a separate FAS file for each source file, or to merge all compiled files into a single FAS file.</p> <p>A single FAS file is faster to load and is required for certain types of optimization. Sometimes, however, you will prefer to load your code one file at a time. This is important if you have not completed the debugging or modification of the application’s code. FAS files do not allow source code debugging, so it is recommended that you compile your code only after the initial debugging is done.</p>
Edit Global Declarations	<p>Create or edit a global declarations file for the project.</p> <p>This feature is provided for compatibility with the Preview version of VLISP. The functionality provided by the Global Declarations file will be provided by other features in future releases of VLISP.</p>
FAS Directory	Specify the directory for compiled files. If you indicate a relative path, VLISP applies it in relation to the project’s home directory. If you leave the field blank, VLISP places compiled files in the same directory as the project definition ( <i>.prj</i> ) file.
Tmp Directory	Specify the directory for project-related temporary files. A relative path is applied in relation to the project’s home directory.

Link Mode	<p>Specify how function calls are to be optimized. This option is only available if optimized compilation is selected. Choose from the following:</p> <ul style="list-style-type: none"> <li>■ <b><i>Do not link</i></b>: This results in indirect linking of functions. The compiler stores the address of the symbol naming the function.</li> <li>■ <b><i>Link</i></b>: If selected, the compiler directly addresses function definitions and all calls where the functions are referenced.</li> <li>■ <b><i>Internal</i></b>: This directly links function calls and removes (drops) the function names from the resulting FAS files.</li> </ul> <p>See “Choosing a Link Mode” on page 157 for further information on these options.</p>
Localize Variables	<p>If selected, the compiler removes (drops) the names of all local symbols from compiled files and directly links their references, wherever possible. This means the program code points to the address where a variable is stored, not to a symbol used to find the address of the variable.</p>
Safe Optimize	<p>If selected, this option directs the compiler to refuse some types of optimization, if there is a chance they will result in incorrect code. For more information on optimization, see “Choosing a Compilation Mode,” below.</p>
Message Mode	<p>Select the level of detail you want VLISP to produce in its compilation reports. You can choose to receive a report showing only fatal errors (those causing compilation failure), a report showing errors and warning messages, or a full report showing errors, warnings, and compiler statistics.</p>

## Choosing a Compilation Mode

Combining compiled code from multiple files to a single binary file allows the compiler to add a high level of optimization. It also means that there are more choices to make.

When producing standard, non-optimized binary code, the VLISP compiler preserves the symbol names associated with functions and global variables, because these symbols may be referenced from other files. When the symbol is referenced, VLISP looks in a table to determine what area in memory is assigned to the symbol.

When optimizing code, the VLISP compiler assumes all files in a project work together to form a complete application. This allows the compiler to discard the symbol names and, when executing the code, jump directly to the memory location containing the value associated with the symbol.

## Analyzing for Optimization Correctness

Optimizing code may introduce bugs to software that runs perfectly when non-optimized. Also, the level of performance gain depends highly on the internal structure of the source code. LISP is a language in which you can easily write programs that create or modify functions at runtime. This use of the language by definition contradicts compile-time optimization.

The VLISP compiler analyzes the code it compiles and links, then it creates a report pointing you to all source code segments that may cause problems when optimized. If you do not receive any optimization warning messages, you can assume optimization did not introduce new problems to your code.

The compiler is able to detect most problematic situations in AutoLISP code. However, there are situations in which it is impossible to detect code that may become incorrect during the optimization. If your program uses one of the following constructs, the compiler will not be able to prove correctness of the optimized code definitively:

- Interaction with external ObjectARX applications that set or retrieve AutoLISP variables
- Dynamic calls to functions defined by other ObjectARX applications
- Evaluation of dynamically built code using `eval`, `apply`, `mapcar`, or `load`
- Use of `set` to set dynamically supplied variables
- Dynamic (program evaluated) action strings in `action_tile` and `new_dialog`

Remember that any optimization will change program semantics. The compiler intends to preserve the behavior of project components relative to one another. The compiler cannot guarantee unchanged behavior between your project and external procedures. Typical effects of optimization include the following:

- Outer applications and the VLISP Console window lose access to program functions and symbols.
- Functions available from the Console window in interpreter mode are unknown in compiled mode.
- Functions are available from the Console window, but redefining them does not change the program's behavior.



## Choosing a Link Mode

If you instruct the VLISP compiler to link functions in your project directly, the compiler tries to resolve all explicit function calls by referencing the function's definition in memory. In contrast, when you indirectly link your functions, the compiler creates references to symbols that VLISP later uses to look up the actual memory location of the function. Direct linking improves the performance of the compiled code and protects the code against function redefinition. However, if your application needs to redefine a function, you cannot directly link that function.

Once function calls are directly linked, the compiler can optimize one level further by dropping the function name completely so that the function becomes invisible to users. To select this feature, choose the Internal Link mode option. Note that symbols exported to ACAD (for example, function names starting with C: ) are never dropped.

## Understanding Safe Optimization

Choosing the Safe Optimize option reduces the amount of compiler optimization but protects your code against compiler-induced errors. Safe optimizing prevents runtime uncertainty that could cause an optimized program to fail, even though the source code seems to be correct. For example, imagine the following situation:

- The function symbol `fi shl i ps` is defined by `defun` and used somewhere in your code. This is a typical candidate for link optimization.
- In another segment of your code, a variable named `fi shl i ps` is assigned using `(setq fi shl i ps expression)`.

Now there are two possible conditions. If the value assigned through `setq` is intended to alter the definition of the function `fi shl i ps`, direct linking will prevent this from happening. The first definition will be referenced directly and cannot be changed by the `setq` function. On the other hand, if the identical names are handled independently, `fi shl i ps` can be linked without creating incorrect code.

If safe optimizing is on, the compiler will always stay on the safe side, even if you explicitly request that `fi shl i ps` be directly linked. This may result in less efficient code, but it ensures code correctness. If safe optimizing is off, you can override the compiler's recommendation to link `fi shl i ps` indirectly. You are responsible for the link option.

The Safe Optimize mode is on by default. Be sure you fully understand the consequences before you turn it off.

## Optimization Conditions Bypassed by Safe Optimization

When Safe Optimization is in effect, the VLISP compiler ignores an optimization option if it determines that adhering to the option may induce an error condition. The following list identifies the conditions under which an option is ignored:

Link	<p>If the compiler encounters the following situations while Safe Optimize is on, it ignores any related Link directive:</p> <ul style="list-style-type: none"><li>■ A symbol is bound as a parameter anywhere in the project.</li><li>■ A symbol is bound as an auxiliary variable and referenced by value anywhere in the project.</li><li>■ A symbol is explicitly assigned somewhere (by setq).</li></ul>
Drop	<p>If the compiler encounters a symbol referenced by value, it ignores any Drop directive for the symbol.</p>
Localize	<p>If the compiler encounters the following situations while safe optimize is on, it ignores the Localize directive or the corresponding variable:</p> <ul style="list-style-type: none"><li>■ A variable has a non-local reference or assignment to it within the project.</li><li>■ A variable is called by name.</li></ul>

## Safe Optimization Warning Messages

If optimized compilation is on and the compiler finds a condition that forbids a certain level of optimization, it issues a warning message. For example, if the function `fi sh l i ps` cannot be linked because the compiler found two definitions for the function, you'll see:

```
; *** WARNI NG: Cannot LINK fi sh l i ps;  
Two DEFUNs found.  
See Another DEFUN
```

Right-click on a warning message to open a shortcut menu. In addition to displaying symbol commands, the menu allows you to view the source code associated with the message. Double-click on the highlighted message to show the source code. To browse all source files related to the compiler messages, press F11 repeatedly, or press SHIFT + F11 to return to the first message.

Each line of the previous warning message guides you to a different code segment. For example:

```
; *** WARNING: Cannot LINK fi shl i ps
```

shows the function call that could not be linked.

```
; Two DEFUNs found
```

shows the first defun found for function fi shl i ps.

```
; See Another DEFUN
```

shows the second defun found for function fi shl i ps.

When the compiler works in Safe Optimization mode and finds a problem condition, the warning starts with:

```
; *** WARNING: Safe: Cannot ...
```

If Safe Optimization is off, but message mode is set to Full report, the same warnings are prefixed by:

```
; *** WARNING: Dangerous ...
```

If you disable Safe Optimize mode, these problematic conditions result in compiler warnings.

## Compiler Checking of Optimizing Conditions

The compiler always checks for optimizing consistency. If you specify an optimization option that contradicts certain security rules, the compiler will issue warning messages. The security rules are:

Link                      The compiler directly links AutoLISP function calls only if the following conditions are met:

- The function is defined only once, or is predefined by AutoLISP and no user defun redefines it.
- The function name does not appear in the parameter list of another function.
- The function is not assigned anywhere in the project.

## Drop

The compiler tries to drop a function symbol only if all corresponding function calls are directly linked to the function definition. The compiler does not drop the symbol for a function definition if the program calls the function by its symbol name. A function is called by symbol in the following cases:

- The symbol appears in a `vl -acad-defun` declaration.
- The function was called from an `ACTION_TILE` action string.
- The function symbol is a quoted argument for `apply`, `mapcar`, or `eval` somewhere in the project.

Note that for functions called from top-level expressions, the Drop declaration will be ignored without warning messages.

## Localize

The compiler does not localize a variable in bound lists of `defun`, `lambda`, and `foreach` expressions if any of the following conditions are true:

- The variable has a non-local reference (or assignment) to it within the outer top-level expression.
- The variable is called as a function by name.
- The variable symbol appears as a function call somewhere in the top-level read-eval loop.

Other than these conditions, which always cancel the optimization and result in warning messages, there are other conditions that may or may not result in incorrect code. Choose the Safe Optimize option for the project to disallow these conditions as well. Disabling Safe Optimization results in compiler warnings if these conditions are met. See “Understanding Safe Optimization” on page 157 for more information on this topic.

# Working with ActiveX

VLISP not only makes program development easier and faster, it also provides new functionality to AutoLISP applications. For example, you can use VLISP to access ActiveX objects from AutoLISP code. You can also use ActiveX to interact with other Windows applications that support ActiveX methodology.

## In This Chapter

# 6

- Using ActiveX Objects with AutoLISP
- Understanding the AutoCAD Object Model
- Accessing AutoCAD Objects
- Using Visual LISP Functions with ActiveX Methods
- Using ActiveX to Interact with Other Applications

# Using ActiveX Objects with AutoLISP

ActiveX Automation is a new way to work programmatically with the contents of an AutoCAD drawing. In many instances, ActiveX works faster than traditional AutoLISP functions in manipulating AutoCAD drawing objects.

The ActiveX programming interface is usable from a number of languages and environments, such as C++, Visual Basic™, and Delphi™. When you work with ActiveX objects in AutoLISP, you work with the same object model, properties, and methods that can be manipulated from other programming environments.

Objects are the main building blocks of an ActiveX application. In some ways, you are already familiar with this notion. For example, AutoCAD drawing items such as lines, arcs, polylines, and circles have long been referred to as objects. But in the ActiveX schema, the following AutoCAD components are also represented as objects:

- Style settings, such as linetypes and dimension styles
- Organizational structures, such as layers, groups, and blocks
- The drawing display, such as the view and viewport
- The drawing's model space and paper space

Even the drawing and the AutoCAD application itself are considered objects.

ActiveX includes much of the functionality provided by standard AutoLISP functions such as `entget`, `entmod`, and `setvar`. Compared to these functions, ActiveX runs faster and provides easier access to object properties. For example, to access the radius of a circle with standard AutoLISP functions, you must use `entget` to obtain a list of entities and `assoc` to find the property you want. You must also know the code number (DXF key value) associated with that property to obtain it with `assoc`, as shown in the following example:

```
(setq radius (cdr (assoc 40 (entget circle-entity))))
```

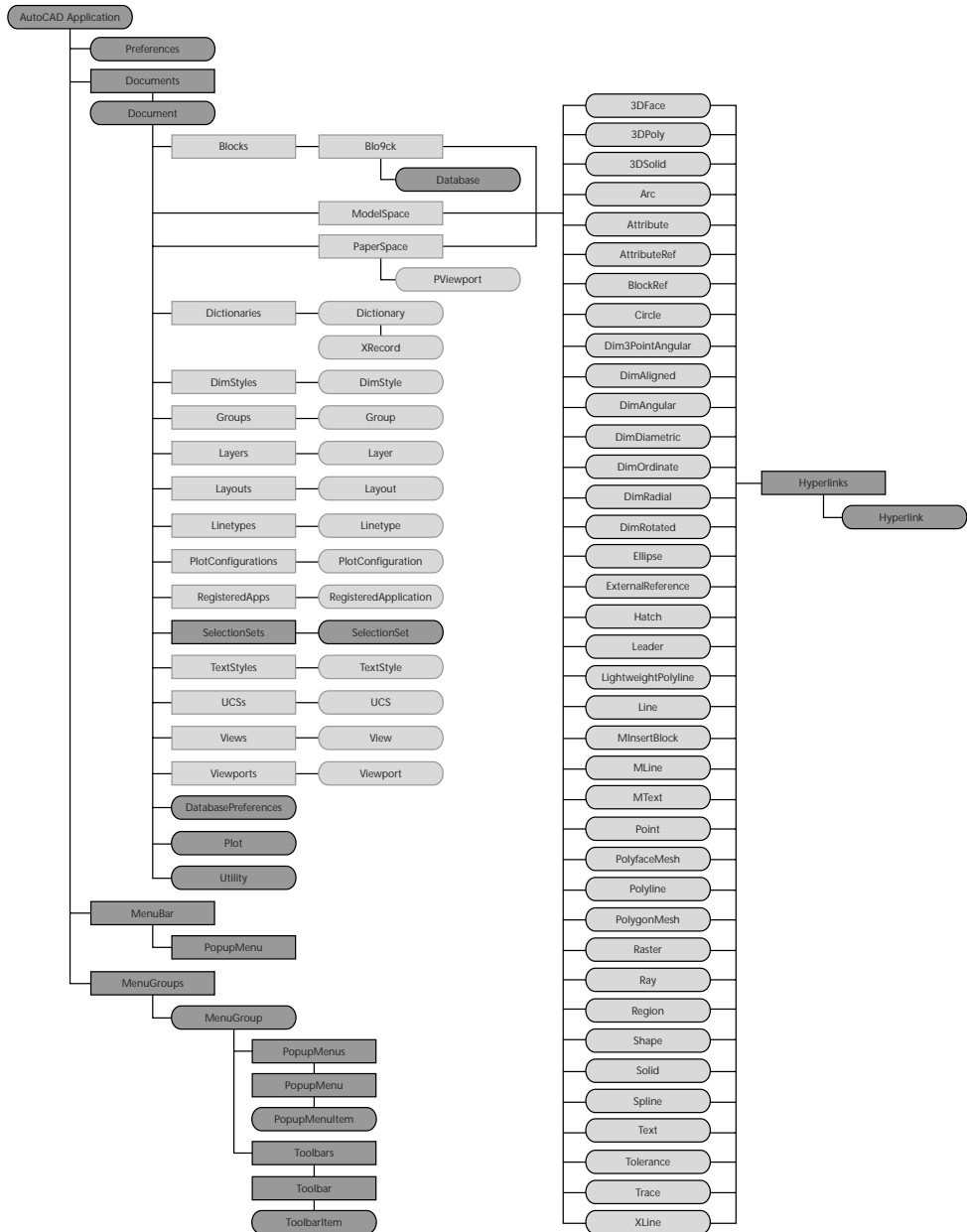
With an ActiveX function, you simply ask for the radius of a circle as follows:

```
(setq radius (vla-get-radius circle-object))
```

## Understanding the AutoCAD Object Model

AutoCAD objects are structured in a hierarchical fashion, with the Application object at the root. The view of this hierarchical structure is referred to as

the object model. It shows you which object provides access to the next level of objects. The AutoCAD object model is described in the following figure:



Using ActiveX is not always a matter of choice. For example, you must use ActiveX to access drawing objects from reactor callback functions. You'll learn more about this in "Attaching Reactors to AutoCAD Drawings" on page 202.

## Object Properties

All objects in the AutoCAD object model have one or more properties. For example, a circle object can be described by properties such as radius, area, or linetype. An ellipse object also has area and linetype properties, but it cannot be described in terms of its radius. Rather, you describe it in terms of its major to minor axis ratio, a property named `RadiusRatio`. Property names are necessary when accessing AutoCAD data through ActiveX functions.

## Object Methods

ActiveX objects also contain methods, which are simply the actions available for a particular kind of object. Some methods can be applied to most AutoCAD drawing objects. For example, the `Mirror` method (creating a mirror image copy of an object around a mirror axis), and the `Move` method (moving a drawing object along a specified vector), can be applied to most drawing objects. By contrast, the `Offset` method, which creates a new object at a specified distance from an existing object, applies only to a few classes of AutoCAD objects such as `Arc`, `Circle`, `Ellipse`, and `Line`.

In VLISP, ActiveX methods are implemented as AutoLISP functions. You'll see many references to ActiveX functions in VLISP documentation, but keep in mind that in ActiveX terminology, they are always known as methods.

To determine which methods and properties apply to a specific type of AutoCAD object, refer to the *ActiveX and VBA Reference*. This reference is available from the VLISP and AutoCAD Help menus, or by opening the *acadauto.hlp* file in the AutoCAD Help directory.

You will probably want to have the *ActiveX and VBA Reference* open at all times when you are developing VLISP programs that use ActiveX. If you open the *acadauto.hlp* file from the AutoCAD Help directory, you can keep the reference open when you use VLISP online help. If you open the reference by selecting it from the AutoCAD Help menu, it closes when you choose an item from the VLISP Help menu.

## Collections of Objects

All objects in the AutoCAD object model are grouped in collections. For example, the `Blocks` collection is made up of all blocks in an AutoCAD draw-



ing, and the ModelSpace collection comprises all graphical objects (circles, lines, polylines, and so on) in the drawing's model space. Collections are labeled in the object model diagram.

## Accessing AutoCAD Objects

The Application object is the root object for the AutoCAD object model. From the Application object, you can access any of the other objects, or the properties or methods assigned to objects.

Before you can use ActiveX functions with AutoLISP, you need to load the supporting code that enables these functions. Issue the following function call to load ActiveX support:

```
(vl -l oad-com)
```

This function first checks if ActiveX support is already loaded; if so, the function does nothing. If ActiveX support is not already loaded, `vl -l oad-com` loads ActiveX and other Visual LISP extensions to the AutoLISP language.

---

**NOTE** All applications that use ActiveX should begin by calling `vl -l oad-com`. If your application does not call `vl -l oad-com`, the application will fail, unless the user has already loaded ActiveX support.

---

After loading the ActiveX support functions, the first step in accessing AutoCAD objects is to establish a connection to the AutoCAD Application object. Use the `vl ax-get-acad-object` function to establish this connection, as in the following example:

```
(setq acadObject (vl ax-get-acad-object))
```

The `vl ax-get-acad-object` function returns a pointer to the AutoCAD Application object. In the example above, the pointer is stored in the *acadObject* variable. This return value exists as a unique VLISP data type called VLA-object (VLISP ActiveX object).

When you refer to AutoCAD objects with ActiveX functions, you must specify a VLA-object type. For this reason, you cannot use `entget` to access an object and then refer to that object with an ActiveX function. The `entget` function returns an object of data type `ename`. Although you cannot use this object directly with an ActiveX function, you can convert it to a VLA-object using the `vl ax-ename->vla-object` function. (See “Converting Object References” on page 190.)

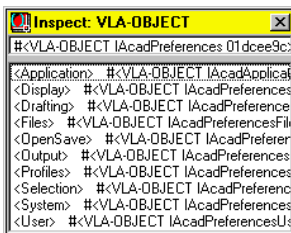
## Using the Inspect Tool to View Object Properties

To view the properties associated with an Application object, you can select the variable that points to the object (*acadobject*, in the previous example), and choose the Inspect button on the VLISP View toolbar as follows:



You can readily identify many of the properties listed in the VLA-object Inspect window. For example, *FullName* is the file name of the AutoCAD executable file, *Version* is the current AutoCAD version, and *Caption* is the contents of the AutoCAD window title bar. An [R0] following a property name indicates the property is read-only; you cannot change it.

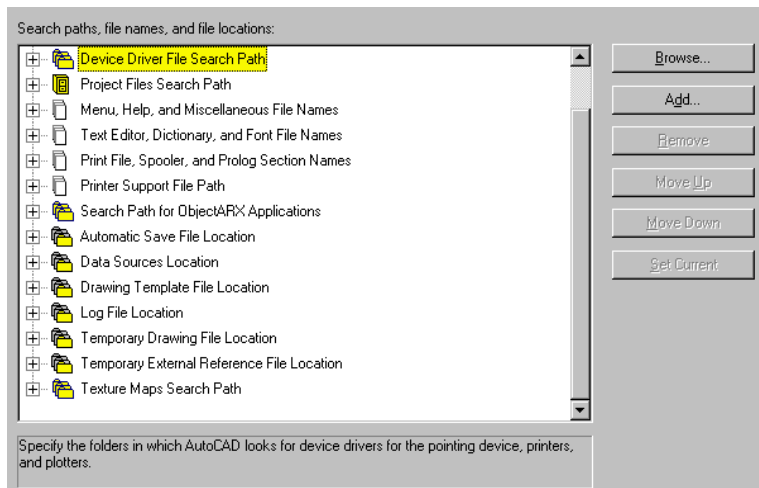
Any property identified as a #<VLA-OBJECT. . . > refers to another AutoCAD ActiveX object. Look at the Preferences property, for example. If you refer to the diagram of the AutoCAD object model, you'll see that the Preferences object is just below the Application object in the model hierarchy. To view the properties associated with an object, double-click the object line in the Inspect window (or right-click and choose Inspect). Here is the Inspect window for the Preferences object:



You may notice that the properties of the Preferences object correspond to the tabs on the AutoCAD Options dialog box. Double-click on the Files property to display the following Inspect window:



If you compare the properties shown in this window to the options available under the Files tab in the AutoCAD Options dialog box, you'll be able to see the connection between the two. The following figure shows the Files options:



For example, the AutoSavePath property corresponds to the Automatic Save File Location option, and the HelpFilePath property would be a sub-option under the Menu, Help, and Miscellaneous File Names option.

You'll learn how to use ActiveX functions to access objects and modify properties in "Using Visual LISP Functions with ActiveX Methods" on page 170.

## Moving Forward from the Application Object

Following the AutoCAD object model hierarchy, the `ActiveDocument` property of the `Application` object leads you to a `Document` object. This `Document` object represents the current AutoCAD drawing. The following AutoLISP command returns the active document:

```
(setq acadDocument (vl a-get-ActiveDocument acadObject))
```

The `Document` object has many properties. Access to non-graphical objects (layers, linetypes, and groups, for example) is provided through like-named properties such as `Layers`, `Linetypes`, and `Groups`. To get to the graphical objects in the AutoCAD drawing, you must access either the drawing's model space (through the `ModelSpace` property) or paper space (through the `PaperSpace` property). For example:

```
(setq mSpace (vl a-get-ModelSpace acadDocument))
```

At this point, you have access to the AutoCAD drawing and can add objects to the drawing. For example, you can add a circle to the model space with the following command:

```
(setq mycircle (vl a-addCircle mSpace  
  (vl ax-3d-point '(3.0 3.0 0.0)) 2.0))
```

## Summarizing the Process

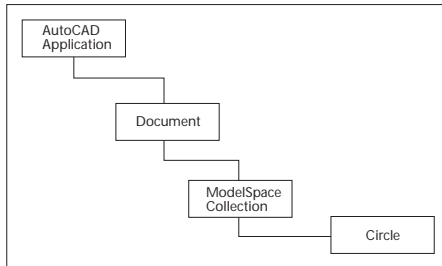
In this section, you saw code examples that led to the drawing of a circle object in an AutoCAD drawing using ActiveX Automation. The following sequence of function calls was used:

```
(vl-load-com)  
(setq acadObject (vl ax-get-acad-object))  
(setq acadDocument (vl a-get-ActiveDocument acadObject))  
(setq mSpace (vl a-get-ModelSpace acadDocument))  
(setq mycircle (vl a-addCircle mSpace  
  (vl ax-3d-point '(3.0 3.0 0.0)) 2.0))
```

The statements in this example accomplished the following:

- Loaded AutoLISP ActiveX support functions.
- Returned a pointer to the `Application` object.
- Obtained a pointer to the current active `Document` object, using the `ActiveDocument` property of the `Application` object. This provided access to the current AutoCAD drawing.
- Obtained a pointer to the `ModelSpace` object, using the `ModelSpace` property of the `Document` object.
- Drew a circle in the `ModelSpace`.

The hierarchical path traversed in the AutoCAD object model is pictured below:



## Performance Considerations

Repeated calls to access the AutoCAD Application, active Document, and Modelspace objects should be avoided, as they negatively impact performance. You should design your applications to obtain these objects one time, and refer to the obtained object pointers throughout the application.

The following code examples illustrate three functions you can define to return the Application, active Document, and Modelspace objects, respectively:

```

(setq *acad-object* nil) ; Initialize global variable
(defun acad-object ()
  (cond (*acad-object*) ; Return the cached object
    (t
     (setq *acad-object* (vlax-get-acad-object))
    )
  )
)

(setq *active-document* nil) ; Initialize global variable
(defun active-document ()
  (cond (*active-document*) ; Return the cached object
    (t
     (setq *active-document* (vla-get-activedocument (acad-object)))
    )
  )
)

(setq *model-space* nil) ; Initialize global variable
(defun model-space ()
  (cond (*model-space*) ; Return the cached object
    (t
     (setq *model-space* (vla-get-modelspace (active-document)))
    )
  )
)

```

For example, you can draw a circle using the following function call:

```
(vl a-addCircle (model-space) (vl ax-3d-point '(3.0 3.0 0.0)) 2.0)
```

The `model-space` function returns the model space of the active document, using the `active-document` function to access the Document object, if necessary. The `active-document` function, in turn, calls `acad-object` to obtain the Application object, if necessary.

## Using Visual LISP Functions with ActiveX Methods

VLISP adds a set of functions to the AutoLISP language that provides access to ActiveX objects. The function names are prefixed with *vla-*: for example, `vl a-addCircle`, `vl a-get-Model Space`, `vl a-getColor`. These functions can be further categorized as follows:

- *vla-* functions correspond to every ActiveX method. Use these functions to invoke the method (for example, `vl a-addCircle` invokes the `Addcircle` method).
- *vla-get-* functions correspond to every property, enabling you to retrieve the value of that property (for example, `vl a-getColor` obtains an object's color property).
- *vla-put-* functions correspond to every property, enabling you to update the value of that property (for example, `vl a-putColor` updates an object's color property).

VLISP also adds a set of ActiveX-related functions whose names are prefixed with *vlax-*. These are more general ActiveX functions, each of which can be applied to numerous methods, objects, or properties. For example, with the `vl ax-get-property` function, you can obtain any property of any ActiveX object. If your drawing contains custom ActiveX objects, or if you need to access objects from other applications, such as a Microsoft Excel spreadsheet, you can use the `vl ax-invoke-method`, `vl ax-get-property`, and `vl ax-put-property` functions to access their methods and properties; you'll see examples using these functions in "Using ActiveX without Importing a Type Library" on page 198.

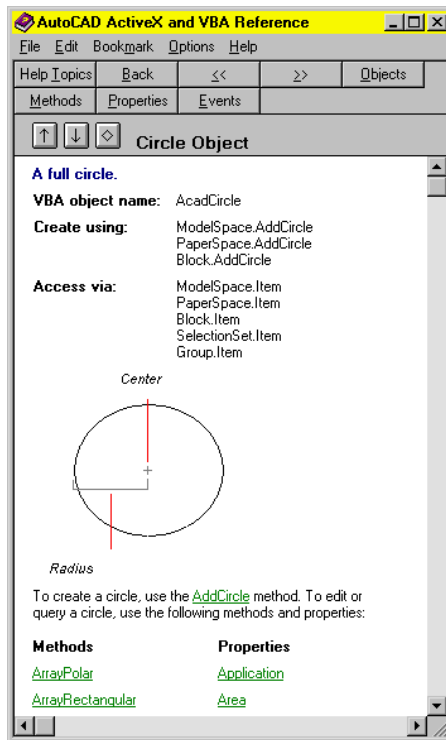
## Determining the Visual LISP Function You Need

The VLISP ActiveX functions actually provide access to ActiveX methods. For example, look at the following AutoLISP statement, which was entered at the VLISP Console prompt:

```
_$ (setq mycircle (vla-addCircle mSpace  
    (vlax-3d-point '(3.0 3.0 0.0)) 2.0))  
#<VLA-OBJECT IAcadCircle 03ad067c>
```

This command adds a circle to a drawing, using the Addcircle method. The function called to draw the circle is `vla-addCircle`.

If you do not know what function adds a circle to an AutoCAD drawing, you can figure it out by looking in the *ActiveX and VBA Reference*. If you look up the definition for a Circle object, here's what the entry looks like:



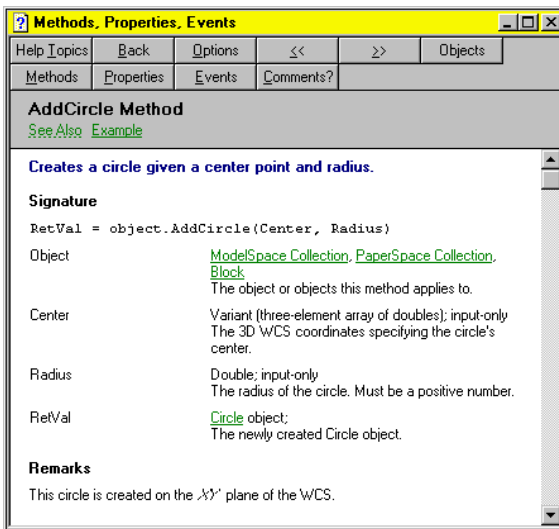
Sometimes, as in this Circle entry, there is descriptive text that identifies the method you need. Often, though, you'll need to look through the list of methods to find the one that matches the action you want to take.

Once you find the name of the method, add a *vla-* prefix to the method name to get the name of the VLISP function that implements the method. In this example, it is `vla-AddCircle`. Note in VLISP the function name is not case-sensitive; `vla-addcircle` is the same as `vla-AddCircle`.

## Determining How to Call a Function

Once you identify the VLISP function you need, you still must determine how to call the function. You need to know the arguments to specify and the data type of those arguments. The *ActiveX and VBA Reference* contains the information required for coding calls to ActiveX functions.

For example, from the reference page for the Circle object, choose the `AddCircle` hyperlink to view the definition of this method:



Note that you can also get to this page by choosing the `Methods` button near the top of the Help window, then choosing `AddCircle` from a list of methods.

The syntax definitions in the reference were designed for Visual Basic users, so they may take some getting used to. For `AddCircle`, the syntax is defined as follows:

```
RetVal = object.AddCircle(Center, Radius)
```

Substituting the variable names used in this chapter's examples, the syntax is:

```
mycircle = mspace.AddCircle(Center, Radius)
```



The AutoLISP syntax required for the same operation is:

```
(setq myCircle (vla-addCircle mSpace  
  (vlax-3d-point '(3.0 3.0 0.0)) 2.0))
```

The return value (RetVal, in Visual Basic) is straightforward. The *ActiveX and VBA Reference* defines this as a Circle object. In VLISP, whenever an AutoCAD object is returned by an ActiveX function, it is stored as a VLA object data type.

The object referred to before the method name (*object.AddCircle*) is always the first argument in a vla function call. This is the AutoCAD object you are viewing or modifying. For example, add a circle to the drawing model space with the following:

```
(vla-addCircle mSpace ...)
```

In this example, *mSpace* refers to the ModelSpace object. Recall from the discussion on the AutoCAD object model (in “Accessing AutoCAD Objects” on page 165), that you use the properties of one AutoCAD object to access another object in a hierarchical manner. The ModelSpace object provides access to the model space of the current drawing.

The *Center* and *Radius* arguments refer to data types that may be unfamiliar to LISP users. The following section explains these data types.

Note that some ActiveX methods require arguments that are described as output only. See “Using ActiveX Methods That Return Values in Arguments” on page 184 for information on how to code these arguments.

## Converting AutoLISP Data Types to ActiveX Data Types

When adding a circle to a drawing, you must specify the center point of the circle and the radius of the circle. In the definition for the AddCircle method in the *ActiveX and VBA Reference*, these arguments are referred to as *Center* and *Radius*. *Center* is defined as a variant (three-element array of doubles), and *Radius* is listed as a double:

```
RetVal = object.AddCircle (Center, Radius)
```

### Elements

Center	Variant (three-element array of doubles); input only. A 3D WCS coordinate specifying the circle's center.
Radius	Double; input only. The radius of the circle. Must be a positive number.

The reference explains what these parameters are used for, but the data types indicated for these parameters may be unfamiliar to LISP users. Variants are essentially self-defining structures that can contain different types of data. For example, strings, integers, and arrays can all be represented by variants. Stored along with the data is information identifying the type of data. This self-defining feature makes variants useful for passing parameters to ActiveX servers, because it enables servers based on any language to understand the data value.

## Working with Variants

Several AutoLISP functions introduced in AutoCAD 2000 allow you to create and work with variants:

- `vlax-make-variant` creates a variant.
- `vlax-variant-type` returns the data type of a variant.
- `vlax-variant-value` returns the value of a variant variable.
- `vlax-variant-change-type` changes the data type of a variant variable.

The `vlax-make-variant` function accepts two arguments: *value* and *type*. The *value* argument is the value to be assigned to the variant. The *type* argument specifies the type of data to be stored in the variant. For *type*, specify one of the following constants:

<code>vlax-vbEmpty</code>	Uninitialized (default value)
<code>vlax-vbNull</code>	Contains no valid data
<code>vlax-vbInteger</code>	Integer
<code>vlax-vbLong</code>	Long integer
<code>vlax-vbSingle</code>	Single-precision floating-point number
<code>vlax-vbDouble</code>	Double-precision floating-point number
<code>vlax-vbString</code>	String
<code>vlax-vbObject</code>	Object
<code>vlax-vbBoolean</code>	Boolean
<code>vlax-vbArray</code>	Array

The constants evaluate to integer values. Because these values may change in future releases of AutoCAD, you should always refer to the constant, not the integer the constant evaluates to. See the entry for `vl ax-make-variant` in the *AutoLISP Reference* for the current integer value assigned to each constant.

For example, the following function call creates an integer variant and sets its value to 5:

```
_$ (setq variant (vl ax-make-variant 5 vl ax-vbl nteger))  
#<variant 2 5>
```

The return value indicates the variant's data type (2, which is `vbl nteger`) and the variant's value (5).

If you do not specify a data type to `vl ax-make-variant`, the function assigns a default type. For example, the following function call creates a variant and assigns it a value of 5 but does not specify a data type:

```
_$ (setq variant (vl ax-make-variant 5))  
#<variant 3 5>
```

By default, `vl ax-make-variant` assigned the specified integer value to a Long Integer data type, not Integer, as you might expect. When assigning a numeric value to a variant, you should explicitly state the data type you want. Refer to `vl ax-make-variant` in the *AutoLISP Reference* for a complete list of default type assignments.

If you do not specify a value or data type, `vl ax-make-variant` allocates an uninitialized (`vl ax-vbEmpty`) variant.

## Working with Safearrays

Arrays passed to ActiveX methods must be of the safearray type. These arrays are safe because you cannot accidentally assign values outside the array bounds and cause a data exception to occur. Use the `vl ax-make-safearray` function to create a safearray and use `vl ax-safearray-put-element` or `vl ax-safearray-fill` to populate a safearray with data.

The `vl ax-make-safearray` function requires a minimum of two arguments. The first argument identifies the type of data that will be stored in the array. Specify one of the following constants for the data type:

<code>vlax-vbInteger</code>	Integer
<code>vlax-vbLong</code>	Long integer
<code>vlax-vbSingle</code>	Single-precision floating-point number
<code>vlax-vbDouble</code>	Double-precision floating-point number
<code>vlax-vbString</code>	String
<code>vlax-vbObject</code>	Object
<code>vlax-vbBoolean</code>	Boolean
<code>vlax-vbVariant</code>	Variant

The constants evaluate to integer values. Because these values may change in future releases of AutoCAD, you should always refer to the constant, not the integer to which the constant evaluates. See the entry for `vl ax-make-safearray` in the *AutoLISP Reference* for the current integer value assigned to each constant.

The remaining arguments to `vl ax-make-safearray` specify the upper and lower bounds of each dimension of the array. You can create single or multi-dimensional arrays with `vl ax-make-safearray`. The lower bound for an index can be zero or any positive or negative integer.

For example, the following function call creates a single-dimension array consisting of doubles, with a starting index of 0:

```
_$ (setq point (vl ax-make-safearray vl ax-vbDouble ' (0 . 2)))  
#<safearray... >
```

The upper bound specified in this example is 2, so the array will hold three elements (element 0, element 1, and element 2).

Different dimensions can have different bounds. For example, the following function call creates a two-dimension array of strings. The first dimension starts at index 0 and contains two elements, while the second dimension starts at index 1 and contains three elements:

```
_$ (setq mat2 (vl ax-make-safearray vl ax-vbString ' (0 . 1) ' (1 . 3)))  
#<safearray... >
```

You can use either `vl ax-safearray-fill` or `vl ax-safearray-put-element` to populate arrays with data.

### Using `vlax-safearray-fill`

The `vl ax-safearray-fill` function requires two arguments: the variable containing the array you are populating and a list of the values to be assigned to the array elements. You must specify as many values as there are elements in the array. For example, the following code populates a single-dimension array of three doubles:

```
(vl ax-safearray-fill point '(100 100 0))
```

You can display the contents of this array in list form with the `vl ax-safearray->list` function:

```
_$ (vl ax-safearray->list point)
(100.0 100.0 0.0)
```

If you do not specify a value for every element in the array, `vl ax-safearray-fill` results in an error.

To assign values to a multi-dimensional array, specify a list of lists to `vl ax-safearray-fill`, with each list corresponding to a dimension. For example, the following command assigns values to a two-dimension array of strings that contains three elements in each dimension:

```
_$ (vl ax-safearray-fill mat2 ' (("a" "b" "c") ("d" "e" "f")))
#<safearray...>
```

Use the `vl ax-safearray->list` function to confirm the contents of `mat2`:

```
_$ (vl ax-safearray->list mat2)
(("a" "b" "c") ("d" "e" "f"))
```

### Using `vlax-safearray-put-element`

The `vl ax-safearray-put-element` function can be used to assign values to one or more elements of a safearray. The number of arguments required by this function depends on the number of dimensions in the array. The following rules apply to specifying arguments to `vl ax-safearray-put-element`:

- The first argument always names the safearray to which you are assigning a value.
- The next set of arguments identifies index values pointing to the element to which you are assigning a value. For a single-dimension array, specify one index value; for a two-dimension array, specify two index values, and so on.
- The final argument is always the value to be assigned to the safearray element.

For example, the following code populates a single-dimension array of three doubles:

```
(vl ax-safearray-put-element point 0 100)
(vl ax-safearray-put-element point 1 100)
(vl ax-safearray-put-element point 2 0)
```

To change the second element of the array to a value of 50, issue the following command:

```
(vl ax-safearray-put-element point 1 50)
```

The following example populates a two-dimension array of strings. The first dimension of the array starts at index 0, while the second dimension starts at index 1:

```
(vl ax-safearray-put-element mat2 0 1 "a")
(vl ax-safearray-put-element mat2 0 2 "b")
(vl ax-safearray-put-element mat2 0 3 "c")
(vl ax-safearray-put-element mat2 1 1 "d")
(vl ax-safearray-put-element mat2 1 2 "e")
(vl ax-safearray-put-element mat2 1 3 "f")
```

You can use `vl ax-safearray->list` to confirm the contents of the array:

```
_$ (vl ax-safearray->list mat2)
(("a" "b" "c") ("d" "e" "f"))
```

## Using Safearrays with Variants

Safearray data must be passed to ActiveX methods through variants. That is, you create a safearray, then you assign the safearray to a variant before passing it to a method. For methods that require you to pass a three-element array of doubles (typically to specify a point), you can use the `vl ax-3d-point` function to build the required data structure. For example, the following call takes a list of points and converts the list into an array of three doubles:

```
_$ (setq ci rcCenter (vl ax-3d-point '(3.0 3.0 0.0)))
#<variant 8197 ... >
```

You can also pass `vl ax-3d-point` two or three numbers, instead of a list. For example:

```
$ (setq ci rcCenter (vl ax-3d-point 3.0 3.0))
#<variant 8197 ... >
```

When you omit the third point from your argument, `vl ax-3d-point` sets it to zero. You can use `vl ax-safearray->list` to verify the contents of the variable set by `vl ax-3d-point`:

```
$ (vl ax-safearray->list (vl ax-variant-value ci rcCenter))
(3.0 3.0 0.0)
```

The `vl ax-TMatrix` function performs a similar task for transformation matrices, which are required by the `vla-TransformBy` function. It builds the

transformation matrix from four lists of four numbers each, converting all numbers to reals, if necessary. For example:

```
_$ (vlax-tmatrix ' ((1 1 1 0) (1 2 3 0) (2 3 4 5) (2 9 8 3)))  
#<variant 8197 ... >
```

If you need to create a variant for an array containing anything other than three doubles or a transformation matrix, you must build it yourself.

To create a variant containing an array of four doubles

- 1 Allocate space for the array:  
(setq 4dubs (vlax-make-safearray vlax-vbDouble ' (0 . 3)))
- 2 Populate the array:  
(vlax-safearray-fill 4dubs ' (3.0 6.0 7.2 1.0))
- 3 Store the safearray in a variant:  
(setq var4dubs (vlax-make-variant 4dubs))

The var4dubs variable now contains a variant containing an array of doubles.

## Converting Other AutoLISP Data Types for ActiveX Methods

The *Radi us* argument to the AddCircle method requires a Double, but the ActiveX functions make the necessary conversion if you specify a real in AutoLISP. The following table identifies the AutoLISP data type that ActiveX functions will accept in place of the required ActiveX data type. Each row in the table represents a data type used by ActiveX functions. Each column in the table represents an AutoLISP data type. Wherever the intersecting cells

contain a plus (+) symbol, you can specify the corresponding AutoLISP data type for the required ActiveX data type.

AutoLISP data types accepted in place of an ActiveX data type							
	Integer	Real	String	VLA-object	Variant	Safe-array	:vlax-true :vlax-false
Byte	+						
Boolean							+
Integer	+						
Long	+						
Single	+	+					
Double	+	+					
Object				+			
String			+				
Variant					+		
Array						+	

In some instances an AutoLISP ActiveX function will accept and convert a data type that is not indicated as acceptable by the preceding table, but you should never count on this.

## Viewing and Updating Object Properties

VLISP provides AutoLISP functions for reading and updating object properties. You can use these functions to obtain the properties of existing drawing objects (for example, the center point of a circle), and to modify drawing objects (for example, moving the center point of the circle).

### Reading Object Properties

Functions that read object properties are named with a *vla-get* prefix and require the following syntax:

```
(vl a-get-property object)
```

For example, `vl a-get-center` returns the center point of a circle.



To obtain an object's property and apply the property to a new object

- 1 Enter the following at the VLISP Console prompt:

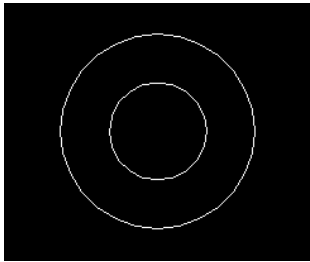
```
(setq myCircle (vla-addcircle mspace (vlax-3d-point  
(getpoint "\nPick the center point for a circle: ")) 2.0))
```

This function call prompts you to pick a center point for a circle, then invokes the Addcircle method to draw the circle. The vlax-3d-point function converts the point you pick into the data type required by vla-addcircle.

- 2 Use vla-get-center to draw a second circle concentric to the first:

```
(vla-addcircle mspace (vla-get-center myCircle) 1.0)
```

The AutoCAD drawing window now contains the following objects:



## Updating Object Properties

Functions that update properties are prefixed with *vla-put* and use the following syntax:

```
(vla-put-property object new-value)
```

For example, vla-put-center changes the center point of a circle.

To change the X axis of a circle

- 1 Obtain the current center point of the circle:

```
_$ (setq myCenter (vla-get-center myCircle))  
#<variant 8197
```

The center point is returned in a variant of type safearray. The safearray contains three doubles (X, Y, and Z coordinates).

- 2 Save the center point in list form:

```
$ (setq centerpt (vlax-safearray->list  
(vlax-variant-value myCenter)))  
(17.8685 5.02781 0.0)
```

Converting the center point from a variant safearray to a list makes it easier to modify the coordinates.

- 3 Subtract 1 from the *X* axis of the center point:

```
_$ (setq newXaxis (- (car centerpt) 1))  
16.8685
```

The result is saved in variable `newXaxis`.

- 4 Construct a new point list for the center point, using the new *X* axis and the original *Y* and *Z* values:

```
_$ (setq newcenter (list newXaxis (cadr centerpt)  
 (caddr centerpt)))  
(16.8685 4.52594 0.0)
```

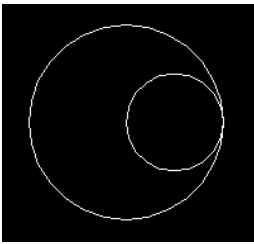
The constructed list is saved in variable `newcenter`.

- 5 Use `vla-put-center` to update the circle with the new *X* axis:

```
_$ (vla-put-center myCircle (vlax-3d-point newcenter))  
nil
```

Note that this command uses `vlax-3d-point` to convert the new center point list into the data type required by `vla-put-center`.

The AutoCAD drawing window shows the result:



Note that changing an object's property may not immediately affect the display of the object in the AutoCAD drawing. AutoCAD delays property changes to allow you to change more than one property at a time. If you need to update the drawing window explicitly, issue the `vla-update` function:

```
(vla-update object)
```

Sometimes you can use pre-defined constants to update an object's property. For example, to set the fill color of a circle to red, you can use the constant `acRed` instead of specifying a numeric index value:

```
(vla-put-color myCircle acRed)
```

The *ActiveX and VBA Reference* lists any predefined constants under the entry describing the property. You can use these constants in VLISP ActiveX function calls.

## Determining Whether an Object Is Available for Updating

If other applications are working with any AutoCAD objects at the same time as your program, those objects may not be accessible. This is especially important to look out for if your application includes reactors, because reactors execute code segments in response to external events that cannot be predicted in advance (see “Attaching Reactors to AutoCAD Drawings” on page 202). Even a simple thing such as a locked layer can prevent you from changing an object’s properties.

VLISP provides the following functions to test the accessibility of an object before trying to use the object:

- `vl ax-read-enabl ed-p` tests whether you can read an object.
- `vl ax-wri te-enabl ed-p` determines whether you can modify an object’s properties.
- `vl ax-erased-p` checks to see if an object has been erased. Erased objects may still exist in the drawing database.

These test functions return `T` if true, `ni l` if false. The following examples test a line object:

Determine whether the line is readable:

```
$ (vl ax-read-enabl ed-p WhatsMyLi ne)  
T
```

Determine whether the line is modifiable:

```
$ (vl ax-wri te-enabl ed-p WhatsMyLi ne)  
T
```

See if the line has been erased:

```
$ (vl ax-erased-p WhatsMyLi ne)  
ni l
```

Erase `WhatsMyLi ne`:

```
_ $ (vl a-del ete WhatsMyLi ne)  
ni l
```

Issue `vl ax-read-enabl ed-p` to see if `WhatsMyLi ne` is still readable:

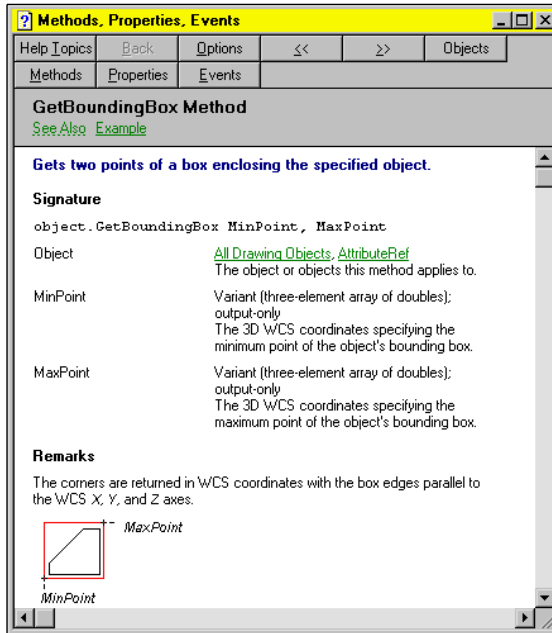
```
$ (vl ax-read-enabl ed-p WhatsMyLi ne)  
ni l
```

Issue `vl ax-erased-p` again to confirm the object was deleted:

```
$ (vl ax-erased-p WhatsMyLi ne)  
T
```

# Using ActiveX Methods That Return Values in Arguments

Some ActiveX methods require that you supply them with variables into which the methods can place values. The `GetBoundingBox` method is an example of this type of method. Here is how it is defined in the *ActiveX and VBA Reference*:



Note that the `MinPoint` and `MaxPoint` parameters are described as output only. You must provide output arguments as quoted variable names. The following example shows a VLISP function call to return the minimum and maximum bounding points of a circle:

```
_$ (vl a-getboundingbox myCircle 'minpoint 'maxpoint)  
nil
```

The values output by `vl a-getboundingbox` are stored in the `minpoint` and `maxpoint` variables as safearrays of three doubles. You can view the values using `vl ax-safearray->list`:

```
_$ (vl ax-safearray->list minpoint)  
(1.0 1.0 -1.0e-008)  
_$ (vl ax-safearray->list maxpoint)  
(5.0 5.0 1.0e-008)
```

Note that the quoted symbol parameters you pass to the function become AutoLISP variables just like the ones created through `setq`. Because of this, you should include them as local variables in your function definition so they do not become global variables by default.

## Listing an Object's Properties and Methods

Earlier in this chapter, you learned how to use the VLISP Inspect tool to display an object's properties. Another way to view an object's properties is to call the `vlax-dump-object` function. You can invoke this function from the VLISP Console window or from an application program. The `vlax-dump-object` function prints a list of the properties of the specified object and returns T. For example, the following code obtains the last object added to the model space, then issues `vlax-dump-object` to print the object's properties:

```
_$ (setq WhatsMyLine (vla-item mSpace (- (vla-get-count mspace) 1)))
#<VLA-OBJECT IAcadLWPolyline 036f1d0c>

_$ (vlax-dump-object WhatsMyLine)
; IAcadLWPolyline: AutoCAD Lightweight Polyline Interface
; Property values:
; Application (R0) = #<VLA-OBJECT IAcadApplication 00a4ae24>
; Area (R0) = 2.46556
; Closed = 0
; Color = 256
; ConstantWidth = 0.0
; Coordinate = ...Indexed contents not shown...
; Coordinates = (8.49917 7.00155 11.2996 3.73137 14.8 5.74379 ...)
; Database (R0) = #<VLA-OBJECT IAcadDatabase 01e3da44>
; Elevation = 0.0
; Handle (R0) = "53"
; HasExtensionDictionary (R0) = 0
; Hyperlinks (R0) = #<VLA-OBJECT IAcadHyperlinks 01e3d7d4>
; Layer = "0"
; Linetype = "BYLAYER"
; LinetypeGeneration = 0
; LinetypeScale = 1.0
; Lineweight = -1
; Normal = (0.0 0.0 1.0)
; ObjectId (R0) = 28895576
; ObjectName (R0) = "AcDbPolyline"
; PlotStyleName = "ByLayer"
; Thickness = 0.0
; Visible = -1
T
```

There is an optional second argument you can supply to `vlax-dump-object` that causes it to also list all the methods that apply to the object. Simply specify "T" following the object name:

```
(vlax-dump-object WhatsMyLine T)
```

Note that `vl ax-dump-object` displays the information in the window from which you issued the command. However, the function returns `T` to the calling program, not the information displayed in the Command window.

## Determining If a Method or Property Applies to an Object

Trying to use a method that does not apply to the specified object will result in an error. Trying to reference a property that does not apply to an object also results in an error. In instances where you are not sure what applies, use the `vl ax-method-appl i cabl e-p` and `vl ax-property-avai l abl e-p` functions to test the objects. These functions return `T` if the method or property is available for the object, and `ni l` if it is not.

The syntax for `vl ax-method-appl i cabl e-p` is:

```
(vl ax-method-appl i cabl e-p object method)
```

The following command checks to see if the `Copy` method can be applied to the object referenced by `WhatsMyLine`:

```
_$_ (vl ax-method-appl i cabl e-p WhatsMyLine "Copy")  
T
```

The following command determines whether or not the `AddBox` method can be applied to the object:

```
_$_ (vl ax-method-appl i cabl e-p WhatsMyLine "AddBox")  
ni l
```

For `vl ax-property-avai l abl e-p`, the syntax is:

```
(vl ax-property-avai l abl e-p object property [T])
```

For example, the following commands determine if `Color` and `Center` are properties of `WhatsMyLine`:

```
_$_ (vl ax-property-avai l abl e-p WhatsMyLine "Col or")  
T  
_$_ (vl ax-property-avai l abl e-p WhatsMyLine "Center")  
ni l
```

Supplying the optional `"T"` argument to `vl ax-property-avai l abl e-p` changes the meaning of the test. If you supply this argument, the function returns `T` only if the object has the property and the property can be modified. If the object has no such property or the property is read-only, `vl ax-property-avai l abl e-p` returns `ni l`. For example, a circle contains an `Area` property, but you cannot update it. If you check the property without specifying the optional argument, the result is `T`:

```
_$_ (vl ax-property-avai l abl e-p myCi rcl e "area")  
T
```

If you supply the optional argument, the result is nil:

```
_$ (vlax-property-available-p myCircle "area" T)
nil
```

## Working with Collection Objects

The concept of collections was introduced in “Understanding the AutoCAD Object Model.” Recall that all ActiveX objects in the AutoCAD object model are grouped in collections. For example, the Blocks collection is made up of all blocks in an AutoCAD document. VLISP provides functions to help you work with collections of AutoCAD objects. These functions are `vlax-map-collection` and `vlax-for`.

The `vlax-map-collection` function applies a function to every object in a collection. The syntax is:

```
(vlax-map-collection collection-object function)
```

For example, the following command displays all properties of every object in a drawing's model space:

```
$ (vlax-map-collection (vla-get-ModelSpace acadDocument)
'vlax-dump-object)
; IAcadLWPolyline: AutoCAD Lightweight Polyline Interface
; Property values:
;   Application (RO) = #<VLA-OBJECT IAcadApplication 00b3b91c>
;   Area (RO) = 3.67152
;   Closed = -1
;   Color = 256
;   Coordinates = (9.59247 4.44872 9.25814 5.34715 4.1991 5.679 ...)
;   EntityName (RO) = "AcDbPolyline"
;   EntityType (RO) = 24
;   Handle (RO) = "4C"
;   Layer = "0"
;   .
;   .
;   .
;   Thickness = 0.0
;   Visible = -1
```

(Note that the preceding example does not show every property returned by `vlax-dump-object`.)

To evaluate a series of functions with each object in a collection, use `vlax-for`:

```
(vlax-for symbol collection [expressions] ...)
```

Like the `foreach` function, `vlax-for` returns the result of the last expression evaluated inside the `for` loop. Note that modifying the collection (that is, adding or removing members) while iterating through it may cause an error.

The following example defines a function that uses `vl ax-for` to show color statistics for each object in the active drawing:

```
(defun show-Color-Statistics (/ objectColor colorSubList colorList)
  (setq modelSpace (vla-get-ModelSpace
    (vla-get-ActiveDocument (vl ax-get-Acad-Object)))
  )
  )
  (vl ax-for obj modelSpace
    (setq objectColor (vla-get-Color obj))
    (if (setq colorSubList (assoc objectColor colorList))
      (setq colorList
        (subst (cons objectColor (1+(cdr colorSubList)))
          colorSubList
          colorList)
        )
      (setq colorList (cons (cons objectColor 1) colorList))
    )
  )
  (if colorList
    (progn (setq
      colorList (vl-sort colorList
        '(lambda (lst1 lst2) (< (car lst1) (car lst2))))
      )
      (princ "\nColorList = ")
      (princ colorList)
      (foreach subList colorList
        (princ "\nColor ")
        (princ (car subList))
        (princ " is found in ")
        (princ (setq count (cdr subList)))
        (princ " object")
        (princ (if (= count 1)
          " "
          "s. ")
        )
      )
    )
    (princ)
  )
)
```

This function lists each color in the drawing and the number of objects where the color is found.

## Retrieving Member Objects in a Collection

The `Item` method retrieves a member object from a collection. The `Count` property shows the number of items in a collection. Using the `Item` method and `Count` property, you can individually process each object in a collection. For example, you can look at each object in a model space, determine the



type of object, and process only the types of objects you are interested in. The following code prints the start angle for each arc object in a model space:

```
(setq index 0)
(repeat (vla-get-count mspace)
  (if (= "AcDbArc" (vla-get-objectname (vla-item mspace index)))
    (progn
      (princ "\nThe start angle of the arc is ")
      (princ (vla-get-startangle (vla-item mspace index)))
    )
  )
  (setq index (+ index 1)))
)
```

Note that `Item` and `Count` also apply to groups and selection sets.

## Releasing Objects and Freeing Memory

Just as you can have different variables pointing to the same AutoCAD entity, you can have multiple VLA-objects pointing to the same drawing object. You can compare two VLA-objects with the `equal` function, which returns `T` if both objects point to the same drawing object.

As long as a VLA-object points to a drawing object, AutoCAD will keep all the memory needed for the object. When you no longer need to reference an object, use the `vlax-release-object` function to indicate this to AutoCAD:

```
(vlax-release-object object)
```

After releasing an object, it is no longer accessible through the VLA-object pointer. This is similar to closing a file. No memory is necessarily freed when you issue `vlax-release-object`, but AutoCAD can reclaim the memory if needed, once all references to the object have been released.

To test whether or not an object has been released, use the `vlax-object-released-p` function:

```
(vlax-object-released-p object)
```

This function returns `T` if the object has been released, `nil` if it has not.

## Converting Object References

There are a number of ways to refer to AutoCAD drawing objects with AutoLISP. These include the following:

- VLA-objects, returned by ActiveX functions
- Entity names (enames), returned by `entget` and `entsel`, identifying objects in an open drawing
- Handles, returned by `handent`, which entities retain across AutoCAD sessions
- Object IDs, used by ARX to identify objects

AutoLISP provides functions to convert from one type of object identifier to another.

### Converting between Enames and VLA-objects

The `vl ax-ename->vl a-object` function allows you to convert entity names (enames) obtained through functions, such as `entget`, to VLA-objects you can use with ActiveX functions. For example, the following code sets a variable to an ename, then uses `vl ax-ename->vl a-object` to convert the ename to a VLA-object:

```
_$ (setq ename-circle (car (entsel "\nPick a Circle:")))
<Entity name: 27f0538>
_$ (setq vlaobject-circle (vl ax-ename->vl a-object ename-circle))
#<VLA-OBJECT IAcadCircle 03642c24>
```

To convert VLA-objects to enames, use `vl ax-vl a-object->ename`. For example:

```
$ (setq new-ename-circle (vl ax-vl a-object->ename vlaobject-circle))
<Entity name: 27f0538>
```

### Obtaining One Object Identifier from Another

You may find the same drawing object represented by different identifiers and data types such as a handle string, an ename, a VLA-object, or an ARX object ID integer. To obtain the identifier with the data type your program requires, use the following strategies:

- To find the handle associated with an ename, use the DXF 5 group of the ename's association list:

```
_$ (setq handle-circle (cdr (assoc 5 (entget ename-circle))))
"4F"
```
- To find the ename associated with a handle, use the `handent` function:

```
_$ (handent handle-circle)
<Entity name: 27f0538>
```

- To find the VLA-object associated with a handle, use the `vl a-handleToObj ect` function:  

```
$ (setq vl a-ci rcl e (vl a-handleToObj ect acadDocument  
  handle-ci rcl e))  
#<VLA-OBJECT IAcadCi rcl e 03642c24>
```
- To find the handle associated with a VLA-object, use `vl a-get-handle` to obtain the handle property:  

```
$ (vl a-get-handle vl a-ci rcl e)  
"4F"
```
- To find the ARX Object ID of a VLA-object, use `vl a-get-objectid` to get the `objectId` property:  

```
_ $ (setq obj id-Ci rcl e (vl a-get-objectid vl a-ci rcl e))  
41878840
```
- To find the VLA-object identified by an ARX Object ID, use the `ObjectIdToObject` method on the AutoCAD Document object:  

```
_ $ (vl a-Obj ectI DtoObj ect acadDocument obj id-ci rcl e)  
#<VLA-OBJECT IAcadCi rcl e 03642c24>
```

## Handling Errors Returned by ActiveX Methods

When ActiveX methods fail, they raise exceptions rather than returning error values that your program can interpret. If your program uses ActiveX methods, you must prepare it to catch exceptions, otherwise the program halts, leaving the user at a Command prompt. You can use `vl -catch-all -apply` to intercept errors returned by ActiveX methods.

### To intercept errors returned by ActiveX methods

- 1 Load the following function and invoke it by issuing `(init-motivate)` at the VLISP Console prompt:

```
(defun init-motivate ()  
  (vl -load-com)  
  (setq mspace  
    (vl a-get-model space  
      (vl a-get-activedocument (vl ax-get-acad-object))  
    )  
  )  
  (vl a-addray mspace (vl ax-3d-point 0 0 0) (vl ax-3d-point 1 1 0))  
)
```

This function adds a ray object to the current model space. A ray has a finite starting point and extends to infinity.

- 2 The `GetBoundingBox` method obtains two points of a box enclosing a specified object, returning those points in variables you supply to the method. (See “Using ActiveX Methods That Return Values in Arguments” on page 184

for an example using this.) The following code obtains a pointer to the last object added to a drawing's model space and uses `vla-getboundingbox` to obtain the points enclosing the object:

```
(defun bnddrop (/ bbox)
  (setq bbox (vla-getboundingbox
              (vla-item mspace (- 1 (vla-get-count mspace)))
              'll
              'ur
              ))
  )
  (list "Do something with bounding box." bbox)
)
```

Load this code and run it by issuing `(bnddrop)` at the Console prompt. Because a ray extends to infinity, it is not possible to enclose it with a box, and `GetBoundingBox` results in the following error:

```
; error: Automation Error. Invalid extents
```

If this code were part of your application program, execution would halt at this point.

- 3 By invoking `vla-getboundingbox` through the `vl-catch-all-apply` function, you can intercept errors returned by ActiveX. Load the following code and run it by issuing `(bndcatch)` at the Console prompt:

```
(defun bndcatch (/ bbox)
  (setq bbox (vl-catch-all-apply
              'vla-getboundingbox
              (list (vla-item mspace (- 1 (vla-get-count mspace)))
                  'll
                  'ur
                  ))
  )
  )
  (if (vl-catch-all-error-p bbox)
      (list "Exception: " (vl-catch-all-error-message bbox))
      (list "Do something with bounding box." bbox)
  )
)
```

This function uses `vl-catch-all-apply` to call `vla-getboundingbox`. It passes `vl-catch-all-apply` two arguments: the symbol naming the function being called (`'vla-getboundingbox`) and a list of arguments to be passed to `vla-getboundingbox`. If the `GetBoundingBox` method completes successfully, `vl-catch-all-apply` stores the return value in variable `bbox`. If the call is unsuccessful, `vl-catch-all-apply` stores an error object in `bbox`.

At this point in the `bnddrop` function, `vla-getboundingbox` was issued directly, an error resulted, and execution halted. But in `bndcatch`, `vl-catch-all-apply` intercepts the error and program execution continues.

A call to `vl-catch-all-error-p` checks the return value from `vl-catch-all-apply` and returns `T` if it is an error object, `nil` otherwise. If the return value is an error object, as it would be in this example, the function issues `vl-catch-all-error-message` to obtain the message from the error object. Program execution continues from this point.

“Catching Errors and Continuing Program Execution” on page 257 includes a non-ActiveX example that uses the `vl-catch-*` functions to intercept errors. For additional information on these functions, see the *AutoLISP Reference*.

## Using ActiveX to Interact with Other Applications

The power of ActiveX extends beyond the ability to interact with standard AutoCAD objects. Visual LISP provides AutoLISP functions that allow you to access objects from other applications that support ActiveX. For example, you can open a Microsoft® Word document, retrieve text data from an AutoCAD drawing, and copy the text into the Word document. Or you might access cells in a Microsoft® Excel spreadsheet and use the data with your AutoCAD drawing.

To write AutoLISP code that interacts with other ActiveX applications, you'll need to refer to the documentation for those applications to learn the application's object names and how to work with its methods and properties. Typically, the online help for an ActiveX-enabled Windows application contains information on its ActiveX interface. For example, AutoCAD provides the *ActiveX and VBA Reference*, as well as the *ActiveX and VBA Developer's Guide* for working with ActiveX using Visual Basic for Applications™ (VBA).

The following topics in this section apply when you work with any ActiveX application from AutoLISP. The code examples illustrate the process of obtaining text from an AutoCAD drawing and then inserting the text into a Microsoft Word document.

### Importing a Type Library

VLISP provides an AutoLISP function that allows you to import the type library of the ActiveX application you want to access. When you import a type library, AutoCAD creates a set of wrapper functions that provide access to the application's methods and properties. In fact, the `vla-` functions you have seen so far are wrapper functions created for the AutoCAD type library.

Use the `vlax-import-type-library` function to import a type library. When calling this function, identify the type library and tell AutoCAD what prefixes to use in naming the wrapper functions for the application's methods and properties. Also specify a prefix for the application's constants. The `vlax-import-type-library` function takes the following syntax:

```
(vlax-import-type-library :tlb-filename filename
 [ :methods-prefix mprefix :properties-prefix pprefix :constants-
 prefix cprefix])
```

The *filename* argument is a string that names the type library. If you do not specify a path, AutoCAD looks for the file in the Support File Search Path.

The *mprefix* argument specifies the prefix to be used for method wrapper functions. For example, if the type library contains a Calculate method and the *mprefix* parameter is set to "cc-", AutoCAD generates a wrapper function named cc-Calculate. This parameter defaults to "".

The *pprefix* argument specifies the prefix to be used for property wrapper functions, and the *cprefix* argument defines the prefix to be used for constants contained in the type library. These parameters also default to "".

Note the required use of keywords when passing arguments to `vlax-import-type-library`. For example, the following code imports a Microsoft Word type library, assigning the prefix `mswm-` to methods, `mswp-` to properties, and `mswc-` to constants:

```
(if (equal nil mswc-wd100Words) ; check for a WinWord constant
  (vlax-import-type-library
    :tlb-filename "c:/Microsoft Office/Office/msword8.olb"
    :methods-prefix "mswm-"
    :properties-prefix "mswp-"
    :constants-prefix "mswc-"
  )
)
```

After importing the type library you can use the VLISP Apropos feature to list the ActiveX wrapper functions resulting from the import. For example, enter `mswm` in the Apropos Options dialog box and select the Match by Prefix option to list all Microsoft Word ActiveX methods.

Importing an application's type library enables you to use VLISP features such as Apropos on the application's properties and methods, but you can access the application even if you do not import its type library. See "Using ActiveX without Importing a Type Library" on page 198.

## Coding Hints for Using `vla-import-type-library`

In AutoCAD 2000, `vla-import-type-library` is executed at runtime rather than at compiletime. In future AutoCAD releases, this may change. The following practices are recommended when using `vla-import-type-library`:

- If you want your code to run on different machines, avoid specifying an absolute path in the *tlb-filename* parameter.
- If possible, avoid using `vla-import-type-library` from inside any AutoLISP expression (in other words, always call it from a top-level position).
- In your AutoLISP source file, code the `vla-import-type-library` call before any code that uses method or property wrappers or constants defined in the type library.

## Establishing a Connection to an Application

If an instance of Microsoft Word is already running on your PC, use `vla-get-object` to establish a connection to the application. For example, the following function call establishes a connection to a Microsoft Word 97 application, and saves a pointer to the application in a variable named `msw`:

```
(setq msw (vla-get-object "Word.Application.8"))
```

The `vla-create-object` function creates a new instance of an application object. For example, if the return value from `vla-get-object` is `nil`, indicating that the requested application does not exist, you can use `vla-create-object` to start the application. The following call starts Microsoft Word 97 and saves a pointer to the application in variable `msw`:

```
(setq msw (vla-create-object "Word.Application.8"))
```

Alternatively, you can use `vla-get-or-create-object` to access an application. This function attempts to connect to an existing instance of an application, and starts a new instance if it doesn't find one.

The application object does not appear until you make it visible. You make an object visible by setting its `Visible` property to `TRUE`. For example, the following call makes the Microsoft Word application visible:

```
(vla-put-visible msw :vla-true)
```

## Coding a Sample Application

After accessing the application object, the remaining AutoLISP code is specific to the application you're working with and the tasks you want to accomplish. For example, the following function call accesses the documents collection object in Microsoft Word:

```
(setq docs (vla-get-documents msw))
```

The following command creates a new Word document:

```
(setq doc (mswm-add docs))
```

The following procedure creates an ActiveX application that works with Microsoft Word 97 and an AutoCAD drawing that contains mtext.

### To copy mtext from an AutoCAD drawing into an Microsoft Word document

- 1 Issue the following command to ensure that AutoLISP ActiveX support is loaded:

```
(vl-load-com)
```

- 2 Obtain the AutoCAD application and current ModelSpace objects and save their pointers:

```
(setq *AcadApp* (vla-get-acad-object)) ; Get AutoCAD application
```

```
(setq *ModelSpace* (vla-get-ModelSpace  
  (vla-get-ActiveDocument *AcadApp*))) ; Get model space
```

- 3 Import the Microsoft Word type library. Change the *tlb-filename* argument in the following code to point to the *msword8.olb* file on your system, then run the code:

```
(if (equal nil mswc-wd100Words) ; check for a Word constant  
  (vla-import-type-library  
    : tlb-filename "c:/Microsoft Office/Office/msword8.olb"  
    : methods-prefix "mswm-"  
    : properties-prefix "mswp-"  
    : constants-prefix "mswc-"  
  )  
)
```

This code first checks to see if a known Microsoft Word constant is defined with a value. If the constant has a value, it is assumed that the Word type library has already been imported and no further action is necessary. If the constant is nil, *vla-import-type-library* is invoked.

- 4 Establish a connection to a Microsoft Word application by running the following code:

```
(setq msw (vla-get-object "Word.Application.8"))  
(if (equal nil msw)  
  (progn
```



```

; Word is not running. Start it.
(setq msw (vla-create-object "Word.Application.8"))
(vla-put-visible msw :vla-true)
)
)

```

The code issues `vla-get-object` to establish a connection to a running Microsoft Word application. (In this example, version 8—Word 97—is specified; if the 8 were omitted, any instance of Word would be accepted.) If there is no running instance of Word, `vla-create-object` is issued to start one.

## 5 The remaining code follows. Comments in the code explain the processing.

```

(if (/= nil msw)
  (progn
    ;; Get the document collection object.
    (setq docs (vla-get-documents msw))
    ;; Add a new document
    (setq doc (mswm-add docs))
    ;; Get the paragraphs of the document (to do some formatting)
    (setq paragraphs (mswp-get-paragraphs doc))
    ;; Now iterate through the model space and export any mtext
    ;; every Mtext entity to Word.
    (vla-for ent *Model Space*
      (if (equal (vla-get-objectName ent) "AcDbMText")
        (progn
          ;; Get the following information from the Mtext entity:
          ;;   o the text string
          ;;   o the location of a corner of the text boundary

          (setq text (vla-get-TextString ent)
                textpos (vla-get-InsertionPoint ent)
                arrayTextpos (vla-variant-value textpos)
                textinfo
                  (strcat
                    (rtos (vla-safearray-get-element arrayTextpos 0) 2 2)
                    " "
                    (rtos (vla-safearray-get-element arrayTextpos 1) 2 2)
                    " "
                    (rtos (vla-safearray-get-element arrayTextpos 2) 2 2)
                  )
                )
          ;; _ end ofsetq
          ; Print some info (with formatting)
          ; Get the last paragraph in the document
          (setq pg (mswp-get-last paragraphs))
          ; Obtain the range of the paragraph
          (setq range (mswp-get-range pg))
          ; Do some formatting
          (mswp-put-bold range 1) ; bold
          (mswp-put-underline range mswc-wdUnderlineSingle) ; underline
          ; 4) Insert info about the text at the end of the paragraph
          (mswm-insertAfter range
            (strcat "AcDbMText at position " textinfo "\n"))

```

```

        ; Now show the text string (from the ACAD text entity)
        (setq pg (mswp-get-last paragraphs))
        (setq range (mswp-get-range pg))
        (mswp-put-bold range 0)
        (mswp-put-underline range mswc-wdUnderlineNone)
        (mswm-InsertAfter range (strcat text "\n\n"))
    ) ;_ end of progn
  ) ;_ end of if AcDbMText
) ;_ end of vlax-for
) ;_ end of progn
(princ "\nNo Microsoft Word application found.\n")
)

```

Load and run the code and look at the result in Microsoft Word.

## Using ActiveX without Importing a Type Library

Importing an application type library and using the resulting ActiveX wrapper functions is convenient and provides access to VLISP features such as *Apropos*, but it comes at a cost. Applications such as Microsoft Word and Microsoft Excel contain hundreds of methods and properties, and creating wrappers for each of these adds up to significant memory usage. Also, you may need to use an ActiveX property or method for which there is no generated AutoLISP wrapper function. In this instance, and to avoid the overhead involved in importing a type library, VLISP provides the following AutoLISP functions:

- `vlax-invoke-method`
- `vlax-get-property`
- `vlax-put-property`

### Calling an ActiveX Method with `vlax-invoke-method`

The `vlax-invoke-method` function calls an ActiveX method directly. The function requires the following arguments:

- The VLA-object the method is to work on
- A symbol or string naming the method to be called
- One or more arguments to be passed to the method

The return value from `vlax-invoke-method` depends on the method being invoked.

For example, the following invokes the `AddCircle` method to draw a circle in the model space (represented by the `mspace` variable) of the current AutoCAD drawing:

```

_.$ (setq mycircle
      (vlax-invoke-method mspace 'AddCircle circleCenter 3.0))
#<VLA-OBJECT IAcadCircle 00bfd6e4>

```

In this example, `ci rcCenter` is a variant (containing a three-element array of doubles) identifying the center of the circle, and 3.0 is the radius of the circle. The method returns a VLA-object, the circle drawn.

## Obtaining an ActiveX Property with `vla-get-property`

The `vla-get-property` function returns the property of an object. The function requires the following arguments:

- A VLA-object identifying the object whose property you are interested in
- A symbol or string naming the property to be retrieved

For example, there is no wrapper function available to obtain the `CommandBars` property of an Microsoft Word application object, but the following command achieves this:

```
_$ (setq ComBars (vla-get-property msw 'CommandBars))  
#<VLA-OBJECT CommandBars 0016763C>
```

You can use `vla-get-property` (and `vla-invoke-method` and `vla-put-property`) even if a wrapper function is available for the task. For example, the following returns the AutoCAD's `ActiveDocument` property:

```
_$ (vla-get-property acadObject 'ActiveDocument)  
#<VLA-OBJECT IAcadDocument 00302a18>
```

In this instance, you could have instead used `vla-get-ActiveDocument` to obtain the `ActiveDocument` property.

## Updating an ActiveX Property with `vla-put-property`

The `vla-put-property` function updates the property of an object. The function requires the following arguments:

- A VLA-object identifying the object whose property you are updating
- A symbol or string naming the property to be set
- The value to set the property to

If `vla-put-property` updates the property successfully, it returns `nil`.

The following function call changes the size of the Microsoft Word toolbar buttons by updating the `LargeButtons` property of the `CommandBars` object:

```
_$ (vla-put-property combars 'LargeButtons :vla-true)  
nil
```



# Advanced Topics

VLISP not only makes program development easier and faster, it also provides new functionality to AutoLISP applications. For example, you can attach reactors to entities in the AutoCAD drawing window, allowing your application to respond to user actions on these entities.

## In This Chapter

# 7

- Attaching Reactors to AutoCAD Drawings

# Attaching Reactors to AutoCAD Drawings

A reactor is an object you attach to AutoCAD drawing objects to have AutoCAD notify your application when events you are interested in occur. For example, if a user moves an entity that your application has attached a reactor to, your application will receive notification that the entity has moved. If you design it to do so, your application can react to this notification with appropriate actions, such as moving other entities associated with the one moved, or perhaps updating a text tag that records revision information on the altered drawing feature.

A reactor communicates with your application by calling a function you have associated with the reactor. Such a function is referred to as a callback function. There isn't anything particularly unusual about reactor callback functions—they are like other functions you write with VLISP. They become callback functions when you attach them to reactor events.

Before you can use reactor functions with AutoLISP, you must load the supporting code that enables these functions. Issue the following function call to load reactor support:

```
vl-load-com
```

This function first checks whether reactor support is already loaded; if reactor support is loaded, the function does nothing, otherwise, it loads reactor support and other AutoLISP extended functions.

---

**NOTE** All applications that use reactors—including all callback functions—should begin by calling `vl-load-com`.

---

## Understanding Reactor Types and Events

There are many types of AutoCAD reactors. Each reactor type responds to one or more AutoCAD events. The different types of reactors are grouped into the following categories:

Database Reactors	Database reactors notify your application when specific events occur to the drawing database, such as when an object has been added to the database.
Document Reactors	Document reactors notify your application of a change to the current drawing document, such as opening a new drawing document, activating a different document window, and changing a document's lock status.

Editor Reactors	Editor reactors notify you each time an AutoCAD command is invoked; a drawing opens, closes, or is saved; a DXF file is imported or exported; or a system variable changes value.
Linker Reactors	Linker reactors notify your application every time an ARX application is loaded or unloaded.
Object Reactors	Object reactors notify you each time a specific object is changed, copied, or deleted.

With the exception of Editor reactors, there is one type of reactor for each reactor category. The following table lists the name by which each reactor type is identified in AutoLISP code:

General reactor types	
Reactor type identifier	Description
:VLR-AcDb-Reactor	Database reactor
:VLR-DocManager-Reactor	Document management reactor
:VLR-Editor-Reactor	General Editor reactor—maintained for backward-compatibility
:VLR-Linker-Reactor	Linker reactor
:VLR-Object-Reactor	Object reactor

Beginning with AutoCAD 2000, the broad class of Editor reactors is broken down into more specific reactor types. The :VLR-Editor-Reactor type is retained for backward-compatibility, but any new Editor reactors introduced with AutoCAD 2000 cannot be referenced through :VLR-Editor-Reactor. The following table lists the types of Editor reactors available beginning with AutoCAD 2000.

Editor reactor types	
Reactor type	Description
:VLR-Command-Reactor	Notifies of a command event
:VLR-DeepClone-Reactor	Notifies of a deep clone event
:VLR-DWG-Reactor	Notifies of a drawing event (for example, opening or closing a drawing file)

### Editor reactor types (continued)

Reactor type	Description
:VLR-DXF-Reactor	Notifies of an event related to reading or writing of a DXF file
:VLR-Insert-Reactor	Notifies of an event related to block insertion
:VLR-Lisp-Reactor	Notifies of a LISP event
:VLR-Miscellaneous-Reactor	Does not fall under any of the other editor reactor types
:VLR-Mouse-Reactor	Notifies of a mouse event (for example, a double-click)
:VLR-SysVar-Reactor	Notifies of a change to a system variable
:VLR-Toolbar-Reactor	Notifies of a change to the bitmaps in a toolbar
:VLR-Undo-Reactor	Notifies of an undo event
:VLR-Wblock-Reactor	Notifies of an event related to writing a block
:VLR-Window-Reactor	Notifies of an event related to moving or sizing an AutoCAD window
:VLR-XREF-Reactor	Notifies of an event related to attaching or modifying XREFs

Use the `vl r-types` function to return the complete list of reactor types.

## Reactor Callback Events

For each reactor type there are a number of events that can cause the reactor to notify your application. These events are known as callback events, because they cause the reactor to call a function you associate with the event. For example, when you issue the Save command to save a drawing, a `:vl r-beginSave` event occurs. When you complete the save process, a `:vl r-saveComplete` event occurs. In designing a reactor-based application, it is up to you to determine the events you are interested in, and to write the callback functions to be triggered when these events occur.

The `vl r-reaction-names` function returns a list of all available events for a given reactor type:

`(vl r-reaction-names reactor type)`



For example, the following command returns a list of all events related to Object reactors:

```
$ (vlr-reaction-names :VLR-Object-Reactor)
(: VLR-cancelled : VLR-copied : VLR-erased : VLR-unerased : VLR-goodbye
: VLR-openedForModify : VLR-modified : VLR-subObj Modified : VLR-
modifyUndone : VLR-modifiedXData : VLR-unappended : VLR-reappended
: VLR-objectClosed)
```

---

**NOTE** If this or any other `vlr-*` command fails with a “no function definition” message, you may have forgotten to call `vl-load-com`, the function that loads AutoLISP reactor support functions.

---

You can print out a list of all available reactor events, sorted by reactor type, by loading and running the following code in VLISP:

```
(defun print-reactors-and-events ()
  (foreach rtype (vlr-types)
    (princ (strcat "\n" (vl-princ-to-string rtype)))
    (foreach rname (vlr-reaction-names rtype)
      (princ (strcat "\n\t" (vl-princ-to-string rname))))))
  (princ))
```

The *AutoLISP Reference* lists each event available for a reactor type. For each reactor type, you can find this information by looking up the description of the function you use to define a reactor of that type. These functions have the same name as the reactor type, minus the leading colon. For example, `vlr-acdb-reactor` creates a database reactor, `vlr-toolbar-reactor` creates a toolbar reactor, and so on.

## Defining Callback Functions

To add reactor functionality to your application, you first need to write a callback function that performs the tasks needed at the time of the reactor event. After you define a callback function, you link the function to an event by creating a reactor object.

A callback function is a regular AutoLISP function, which you define using `defun`. However, there are some restrictions on what you can do in a callback function. You cannot call AutoCAD commands using the `command` function. Also, to access drawing objects, you must use ActiveX functions; `entget` and `entmod` are not allowed inside callback functions. See “Reactor Use Guidelines” on page 216 for more information.

Callback functions for all reactors, other than Object reactors, must be defined to accept two arguments:

- The first argument identifies the Reactor object that called the function.
- The second argument is a list of parameters set by AutoCAD.

The following example shows a function named `saveDrawingInfo`, which displays file path and size information. This function will be attached to a DWG Editor reactor that will fire when an AutoCAD drawing is saved.

```
(defun saveDrawingInfo (calling-reactor commandInfo / dwgname
  fileInfo)
  (vl-load-com)
  (setq dwgname (cadr commandInfo)
        fileInfo (vl-file-size dwgname))
  (alert (strcat "The file size of " dwgname " is "
                (itoa fileInfo) " bytes."))
  )
  (princ)
)
```

In this example, the `calling-reactor` variable identifies the reactor that invoked the function. The function retrieves the drawing name from the `commandInfo` parameter, then uses the `vl-file-size` function to retrieve the size of the drawing. Finally, the function displays the information in an alert box in the AutoCAD window.

The parameters passed to a callback function depend on the type of event associated with the function. For example, `saveDrawingInfo` will be associated with a `saveComplete` event. This event indicates that a Save command has been completed. For `saveComplete` events, AutoCAD passes the callback function a string containing the name of the file the drawing was saved in. On the other hand, a callback function that reacts to changes to system variables (*sysVarChanged* event) receives a parameter list containing the name of a system variable (a string) and a flag indicating if the change was successful. You can find a list of events for each reactor type, and the parameters associated with each event, in the *AutoLISP Reference*. The events are listed under the description of the functions used to define each type of reactor.

AutoCAD comes with two predefined callback functions. You can use these functions when testing your reactors:

- `vlr-beep-reaction` is a simple function that beeps your PC.
- `vlr-trace-reaction` prints a list of arguments to the VLISP Trace window each time a reactor fires this callback function.

## Defining Object Reactor Callback Functions

Unlike other AutoCAD reactors, object reactors are attached to specific AutoCAD entities (objects). When you define an object reactor, you must identify the entity the reactor is to be attached to. So callback functions for object reactors must be defined to accept three arguments:

- The first argument identifies the object that fired the notification.
- The second argument identifies the Reactor object that called the function.
- The third argument is a list of parameters specific to the callback condition.

For example, the following code defines a callback function named `print-radius`. This function can be used to print the radius of a circle:

```
(defun print-radius (notifier-object reactor-object parameter-list)
  (vl-load-com)
  (cond
    (
      (vlax-property-available-p
        notifier-object
        "Radius")
      )
    (princ "The radius is ")
    (princ (vla-get-radius notifier-object))
  )
)
```

Note that the code uses the `vlax-property-available-p` function to verify that the drawing object that notified this function contains a radius property.

## Creating Reactors

You link a callback function to an event when you create a reactor. There is an AutoLISP function for creating each type of reactor. These functions have the same name as the reactor type, minus the leading colon. For example, `vlr-acdb-reactor` creates a database reactor, `vlr-toolbar-reactor` creates a toolbar reactor, and so on. Except for object reactors, the reactor creation functions require the following arguments:

- AutoLISP data to be associated with the Reactor object
- A list of pairs naming the event and the callback function to be associated with that event (*event-name . callback\_function*)

For example, the following command defines a DWG Editor reactor. The reactor will invoke the `saveDrawingInfo` function in response to a user issuing a `Save` command:

```
(vlr-dwg-Reactor nil ' ((:vlr-saveComplete . saveDrawingInfo)))
```

In this example, the first argument is `nil` because there is no application-specific data to attach to this reactor. The second argument is a list consisting of dotted pair lists. Each dotted pair list identifies an event the reactor is to be notified about, and the callback function to be run in response to that event. In this case the reactor is notified of only one event,

```
:vlr-saveComplete.
```

Editor reactors are notified each time the user issues a command, whether through the AutoCAD Command line, a menu, a toolbar, or an AutoLISP program. So, the callback function for this DWG reactor needs to determine precisely what it is responding to. In the current example, `save-drawing-info` simply checks for the `Save` command.

Possible events for each reactor type are listed in the *AutoLISP Reference*. To find the list of events for a reactor, refer to the entry in the *AutoLISP Reference* that describes the function used to create the reactor. For example, to find the list of possible events for a DWG reactor, refer to the entry for `vlr-DWG-reactor`.

All reactor construction functions return a Reactor object.

## Using Object Reactors

Unlike other AutoCAD reactors, object reactors are attached to specific AutoCAD entities (objects). When you define an object reactor, you must identify the entity the reactor is to be attached to. The `vlr-object-reactor` function, which creates object reactors, requires the following arguments:

- A list of VLA-objects identifying the drawing objects that are to fire notifications to the reactor. These objects are referred to as the reactor owners.
- AutoLISP data to be associated with the Reactor object.
- A list of pairs naming the event and the callback function to be associated with that event (*event-name* . *callback\_function*).

---

**WARNING!** You cannot modify an object in a callback function if it is included in the object reactor's owner list. Attempts to do so will generate an error message and can crash AutoCAD.

---

For example, the following statement defines an object reactor with a single owner (the object identified by `myCircle`), then attaches the string “Circle Reactor” to the reactor and tells AutoCAD to invoke the `print-radius` function when a user modifies `myCircle`:

```
(setq circleReactor (vlr-object-reactor (list myCircle)
    "Circle Reactor" '(:vlr-modified . print-radius))))
```

The Reactor object is stored in variable `circleReactor`; you can refer to the reactor using this variable, as described in “Querying, Modifying, and Removing Reactors” on page 211.

When defining a list of owners, you must specify VLA-objects only; Ename objects are not allowed. VLA-objects are required because callback functions can only use ActiveX methods to modify AutoCAD objects, and ActiveX methods require a VLA-object to work on.

Note that, although you cannot use objects obtained through functions such as `entlast` and `entget` with callback reactors, you can convert these Ename objects into VLA-objects using the `vlax-ename->vla-object` function. See the *AutoLISP Reference* for more information on `vlax-ename->vla-object`.

#### To see how an object reactor works

- 1 Load the following code to define a circle object; you will be prompted to draw the circle:

```
(setq myCircle
  ; Prompt for the center point and radius:
  (progn (setq ctrPt
    (getpoint "\nCircle center point: ")
    radius (distance ctrPt
      (getpoint ctrPt "\nRadius: ")
    )
  )
  ; Add a circle to the drawing model space. Nest the function
  ; calls to obtain the path to the current drawing's model
  ; space: AcadObject > ActiveDocument > Model Space
  (vla-addCircle
    (vla-get-Model Space
      (vla-get-ActiveDocument (vlax-get-acad-object))
    )
    (vlax-3d-point ctrPt)
    radius
  )
)
```

This code uses `vla-addCircle` to draw a circle, assigning the return value to variable `myCircle`. The return value is a VLA-object, which contains a pointer to the Circle object drawn.

- 2 Load the `print-radius` callback function shown in “Defining Object Reactor Callback Functions” on page 207.

- 3 Define the reactor with the following command:

```
(setq circleReactor (vlr-object-reactor (list myCircle)
    "Circle Reactor" '(:vlr-modified . print-radius)))
```

- 4 In the AutoCAD drawing window, select the circle and change its size. The `print-radius` function will display a message in the AutoCAD Command window. For example, if you use the `STRETCH` command to enlarge the circle, the message looks like the following:

```
Specify stretch point or [Base point/Copy/Undo/Exit]: The radius is
3.75803
```

## Attaching Data to Reactor Objects

The object reactor creation example in “Using Object Reactors” included a string, “Circle Reactor,” in the call to `vlr-object-reactor`. You do not have to specify any data to be included with the reactor; you can specify `nil` instead. However, an object may have several reactors attached to it. Include an identifying text string, or other data your application can use, to allow you to distinguish among the different reactors attached to an object.

## Working with Reactors in Multiple Namespaces

The current implementation of AutoLISP supports working in one drawing document at a time. Some AutoCAD APIs, such as ObjectARX and VBA, do support the ability of an application to work simultaneously in multiple documents. As a result, an application may modify an open drawing that is not currently active. This is not supported in AutoLISP. (Note that a VLX may run in a separate-namespace from the document it is loaded from, but it is still associated with that document and cannot manipulate objects in another document.)

AutoLISP does provide limited support for reactor callback functions executing in a document that is not active. By default, a reactor callback function will execute only if a notification event occurs when the document it was defined in is the active document. You can alter this behavior using the `vlr-set-notification` function.

To specify that a reactor should execute its callback function even if the document it was defined in is not active (for example, if an application in another namespace triggers an event), issue the following function call:

```
(vlr-set-notification reactor-object 'all-documents)
```

To modify a reactor so it only executes its callback function if an event occurs when the document it was defined in is active, issue the following:

```
(vlr-set-notification reactor-object 'active-document-only)
```

The `vlr-set-notification` function returns the specified reactor object. For example, the following sequence of commands defines a reactor and sets it to respond to events whether or not its associated document is active:

```
_$ (setq circleReactor (vlr-object-reactor (list myCircle)  
"Circle Reactor" '(:vlr-modified . print-radius)))  
#<VLR-Object-Reactor>
```

```
_$ (vlr-set-notification circleReactor 'all-documents)  
#<VLR-Object-Reactor>
```

To determine the notification setting of a reactor, use the `vlr-notification` function. For example:

```
_$ (vlr-notification circleReactor)  
all-documents
```

The `vlr-set-notification` function affects only the specified reactor. All reactors are created with the default notification set to "active-document-only."

---

**WARNING!** If you choose to set a reactor to execute its callback function even if triggered when its document is not active, the callback function should do nothing other than set and read AutoLISP variables. Any other action may cause system instability.

---

## Querying, Modifying, and Removing Reactors

There are various ways to obtain information about reactors. VLISP supplies AutoLISP functions to query reactors, and you can use standard VLISP data inspection tools to view information on them.

To use AutoLISP to list all reactors in a drawing, call the `vlr-reactors` function. The function returns a list of reactor lists. Each reactor list begins with a symbol identifying the reactor type, followed by pointers to each reactor of that type. For example:

```
_$ (vlr-reactors)  
((:VLR-Object-Reactor #<VLR-Object-Reactor>) (:VLR-Editor-Reactor  
#<VLR-Editor-Reactor>))
```

In this example, `vlr-reactors` returned a list containing two lists, one identifying a single object reactor and one identifying a single Editor reactor.

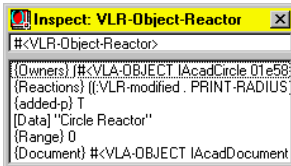
To list all reactors of a given type, supply `vlr-reactors` with an argument identifying the reactor type. Specify one of the values returned by the `vlr-types` function; these are listed in “Understanding Reactor Types and Events” on page 202. For example, the following lists all DWG reactors:

```
_$ (vlr-reactors :vlr-dwg-reactor)
(: VLR-DWG-Reactor #<VLR-DWG-Reactor> #<VLR-DWG-Reactor>))
```

In this case, the return value is a list containing one list. The one list identifies pointers to two DWG reactors.

## Inspecting Reactors

You can examine reactors using the VLISP `Inspect` tool. For example, the object reactor defined in “Using Object Reactors” on page 208 was returned to the variable `circleReactor`. If you open an `Inspect` window for this variable, VLISP displays the following information:



The list items in the `Inspect` window show the following:

- Objects owning the reactor
- Event and associated callback function
- Whether or not the reactor is active (yes if `added-p` is `T`, no if `added-p` is `nil`)
- User data attached to the reactor
- Document range in which the reactor will fire (if 0, it fires only in the context of the drawing document it was created in; if 1, the reactor responds in the context of any document (see “Working with Reactors in Multiple Namespaces” on page 210 for more information on this topic)
- The AutoCAD document attached to the object reactor

Double-click on the item that begins with `{Owners}` to view a list of the owner objects:



Double-click on a list item to obtain detailed information about an owner.



## Querying Reactors Using Function Calls

VLISP also provides functions to inspect a reactor definition from within an application program, or at the Console prompt:

- **vlr-type** returns the type of the specified reactor. For example:  

```
$ (vlr-type circleReactor)  
:VLR-Object-Reactor
```
- **vlr-current-reaction-name** returns the name of the event that caused the callback function to be called.
- **vlr-data** returns the application-specific data value attached to the reactor, as shown in the following example:  

```
$ (vlr-data circleReactor)  
"Circle Reactor"
```

You can use this data to distinguish among multiple reactors that can fire the same callback function.

- **vlr-owners** returns a list of the objects in an AutoCAD drawing that fire notifications to an object reactor. The following function call lists the objects that fire notifications to `circleReactor`:  

```
_$(vlr-owners circleReactor)  
(#<VLA-OBJECT IAcadCircle 03ad077c>)
```
- **vlr-reactions** returns the callback list of condition-function pairs of the specified reactor. The following example returns information about `circleReactor`:  

```
$ (vlr-reactions circleReactor)  
((:vlr-modified . PRINT-RADIUS))
```

## Modifying Reactors

VLISP provides functions to modify reactor definitions:

- **vlr-reaction-set** changes the callback function link for the specified reactor event. The function syntax is:  

```
(vlr-reaction-set reactor callback-condition 'callback-function)
```

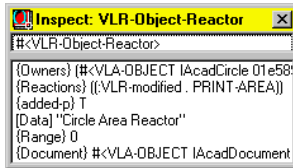
For example, the following command changes the `circleReactor` reactor to call the `print-area` function when an object is modified:

```
$ (vlr-reaction-set circleReactor :vlr-modified 'print-area)  
PRINT-AREA
```
- **vlr-data-set** changes the application-specific data associated with the reactor. For example, the following call replaces the text string used to identify the `circleReactor` reactor:  

```
$ (vlr-data-set circleReactor "Circle Area Reactor")  
"Circle Area Reactor"
```

You can verify that the reactor has changed by using the VLISP Inspect feature. If the Inspect window shown in “Inspecting Reactors” on page 212

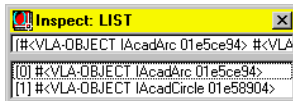
is still displayed in your VLISP session, right-click in the window's object line and choose Update. If you've modified the `circleReactor` reactor as shown in this section, the updated Inspect window will look like the following:



- `vlr-owner-add` adds a database object to the list of owners of the specified reactor. In the following example, an ARC object named `archie` is added to the owner list of reactor `circleReactor`:

```
$ (vlr-owner-add circleReactor archie)
#<VLA-OBJECT IAcadArc 03ad0bcc>
```

Now, if a user modifies the `archie` object, the callback function defined for reactor `circleReactor` is invoked. You can verify this by inspecting the reactor. Update the Inspect window for the `circleReactor` reactor, then right-click on the list item that begins with `{Owners}` and choose Inspect:



Both the Arc and Circle objects are listed in the Inspect window.

- `vlr-owner-remove` removes an Owner object from the list of reactor owners. For example, the following command removes `archie` from the `circleReactor` owner list:

```
$ (vlr-owner-remove circleReactor archie)
#<VLA-OBJECT IAcadArc 03ad0bcc>
```

## Removing Reactors

Use the `vlr-remove` function to disable a reactor. Disabling the reactor does not delete it: the Reactor object still exists and you can activate it again using the `vlr-add` function. To determine whether or not a reactor is active (registered to AutoCAD), use the `vlr-added-p` function:

```
_ $ (vlr-added-p circleReactor)
T
```

The `vlr-added-p` function returns `T` if the reactor is active, `nil` if it is not.

The following command disables reactor `circleReactor`:

```
_ $ (vlr-remove circleReactor)
#<VLR-Object-reactor>
```

You can use `vl r-added-p` to verify the `circleReactor` object reactor has been disabled:

```
$ (vl r-added-p circleReactor)
nil
```

To enable the `circleReactor` reactor, use `vl r-add`:

```
$ (vl r-add circleReactor)
#<VLR-Object-reactor>
```

You can use the `vl r-remove-all` function to disable all reactors in your drawing. To disable all reactors of a particular type, specify the reactor type when calling `vl r-remove-all`. The following function call disables all object reactors:

```
$ (vl r-remove-all :vl r-object-reactor)
(#<VLR-Object-reactor>)
```

## Transient versus Persistent Reactors

Reactors may be transient or persistent. Transient reactors are lost when a drawing closes; this is the default reactor mode. Persistent reactors are saved with the drawing and exist when the drawing is next opened.

Use the `vl r-pers` function to make a reactor persistent. To remove persistence from a reactor and make it transient, use `vl r-pers-release`. Each function takes a Reactor object as its only argument. For example, the following command makes a reactor persistent:

```
_$ (vl r-pers circleReactor)
#<VLR-Object-Reactor>
```

If successful, `vl r-pers` returns the specified Reactor object.

To determine whether a Reactor object is persistent or transient, issue `vl r-pers-p`. For example:

```
_$ (vl r-pers-p circleReactor)
#<VLR-Object-Reactor>
```

The `vl r-pers-p` function returns the Reactor object if it is persistent, `nil` if it is not.

## Opening a Drawing Containing Persistent Reactors

A reactor is only a link between an event and a callback function. While this link remains, the callback function itself is not part of the reactor, and is normally not part of the drawing. The reactors saved in the drawing are only usable if their associated callback functions are loaded in AutoCAD. You can cause this to occur automatically when a drawing is opened if you define the reactor and callback functions in a separate-namespace `VLX`.

If you open a drawing containing VLISP reactor information and the associated callback functions are not loaded, AutoCAD displays an error message. You can use the `vlr-pers-list` function to return a list of all Persistent reactors in a drawing document.

## Reactor Use Guidelines

When using reactors, try to adhere to the following guidelines. Reactors that violate these guidelines can result in unpredictable results for your application if the internal implementation of reactors changes.

- Do not rely on the sequence of reactor notifications.

It is recommended that, with a few exceptions, you do not rely on the sequence of reactor notifications. For example, an OPEN command triggers `BeginCommand`, `BeginOpen`, `EndOpen`, and `EndCommand` events. However, they may not occur in that order. The only event sequence you can safely rely on is that a `Begin` event will occur before the corresponding `End` event. For example, `commandWillStart()` always occurs before `commandEnded()`, and `beginInsert()` always occurs before `endInsert()`. Relying on more complex sequences may result in problems for your application if the sequence is changed as a result of new notifications being introduced in the future and existing ones being rearranged.

- Do not rely on the sequence of function calls between notifications.

It is not guaranteed that certain functions will be called between certain notifications. For example, when you receive a `vlr-erased` notification on object A, all it means is that object A is erased. If you receive a `vlr-erased` notification on A followed by a `vlr-erased` notification on B, all it means is that both objects A and B are erased; it does not ensure that B was erased after A. If you tie your application to this level of detail, there is a very high probability of your application breaking in future releases. Instead of relying on sequences, rely on reactors to indicate the state of the system.

- Do not use any interactive functions in your reactor callback function (for example, do not use `getPoint`, `entsel`).

Attempting to execute interactive functions from within a reactor callback function can cause serious problems, as AutoCAD may still be processing a command at the time the event is triggered. Therefore, avoid the use of input-acquisition methods such as `getPoint`, `entsel`, and `getkeyword`, as well as selection set operations and the `command` function.

- Do not launch a dialog box from within an event handler.

Dialog boxes are considered interactive functions and can interfere with the current operation of AutoCAD. However, message boxes and alert boxes are not considered interactive and can be issued safely.

- Do not update the object that issued the event notification.

The event causing an object to trigger a callback function may still be in progress and the object still in use by AutoCAD when the callback function is invoked. Therefore, do not attempt to update an object from a callback function for the same object. You can, however, safely read information from the object triggering an event. For example, suppose you have a floor filled with tiles and you attach a reactor to the border of the floor. If you change the size of the floor, the reactor callback function will automatically add or subtract tiles to fill the new area. The function will be able to read the new area of the border, but it cannot attempt any changes on the border itself.

- Do not perform any action from a callback function that will trigger the same event.

If you perform an action in your reactor callback function that triggers the same event, you will create an infinite loop. For example, if you attempt to open a drawing from within a BeginOpen event, AutoCAD will simply continue to open more drawings until the maximum number of open drawings is reached.

- Verify that a reactor is not already set before setting it, or you may end up with multiple callbacks on the same event.
- Remember that no events will be fired while AutoCAD is displaying a modal dialog.



# Part 2

## Using the AutoLISP Language



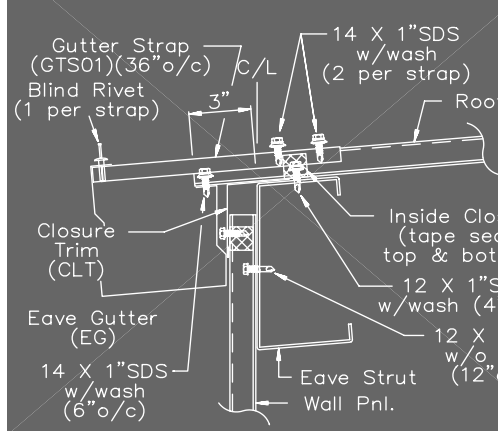




# AutoLISP Basics

This chapter introduces the basic concepts of the AutoLISP programming language. It describes the core components and data types used in AutoLISP, and presents examples of simple number-, string-, output-, and list-handling functions.

AutoLISP code does not need to be compiled, so you can enter the code at a Command line and immediately see the results. Some examples in this chapter are intended to be entered at the Visual LISP (VLISP) Console window prompt (\_\$), while others are entered at the AutoCAD Command prompt (Command:).



## In This Chapter

# 8

- AutoLISP Expressions
- AutoLISP Data Types
- AutoLISP Program Files
- AutoLISP Variables
- Number Handling
- String Handling
- Basic Output Functions
- Equality and Conditional
- List Handling
- Symbol and Function Handling
- Error Handling

# AutoLISP Expressions

An AutoLISP program consists of a series of expressions. AutoLISP expressions have the following form:

```
(function arguments)
```

Each expression begins with an open (left) parenthesis and consists of a function name and optional arguments to that function. Each argument can also be an expression. The expression ends with a right parenthesis. Every expression returns a value that can be used by a surrounding expression. The value of the last interpreted expression is returned to the calling expression.

For example, the following code example involves three functions:

```
(fun1 (fun2 arguments) (fun3 arguments))
```

If you enter this code at the Visual LISP Console prompt or the AutoCAD Command prompt, the AutoCAD AutoLISP interpreter processes the code. The first function, `fun1`, has two arguments, and the other functions, `fun2` and `fun3`, each have one argument. The functions `fun2` and `fun3` are surrounded by function `fun1`, so their return values are passed to `fun1` as arguments. Function `fun1` evaluates the two arguments and returns the value to the window from which you entered the code.

The following example shows the use of the `*` (multiplication) function, which accepts one or more numbers as arguments:

```
_$ (* 2 27)  
54
```

Because this code example has no surrounding expression, AutoLISP returns the result to the window from which you entered the code.

Expressions nested within other expressions return their result to the surrounding expression. The following example uses the result from the `+` (addition) function as one of the arguments for the `*` (multiplication) function.

```
_$ (* 2 (+ 5 10))  
30
```

If you enter the incorrect number of close (right) parentheses, AutoLISP displays the following prompt:

```
(_>
```

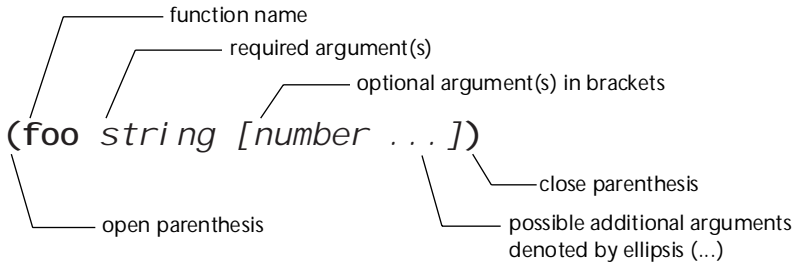
The number of open parentheses in this prompt indicates how many levels of open parentheses remain unclosed. If this prompt appears, you must enter the required number of close parentheses for the expression to be evaluated.

```
_ $ (* 2 (+ 5 10  
(( > ))  
30
```

A common mistake is to omit the closing quotation mark (") in a text string, in which case the close parentheses are interpreted as part of the string and have no effect in resolving the open parentheses. To correct this condition, press SHIFT + ESC to cancel the function, then re-enter it correctly.

## AutoLISP Function Syntax

In this guide, the following conventions describe the syntax for AutoLISP functions:



In this example, the `foo` function has one required argument, `string`, and one optional argument, `number`. Additional `number` arguments can be provided. Frequently, the name of the argument indicates the expected data type. The examples in the following table show both valid and invalid calls to the `foo` function.

Valid and invalid function call examples	
Valid calls	Invalid calls
(foo "catch")	(foo 44 13)
(foo "catch" 22)	(foo "fi " "foe" 44 13)
(foo "catch" 22 31)	(foo)

# AutoLISP Data Types

AutoLISP expressions are processed according to the order and data type of the code within the parentheses. Before you can fully utilize AutoLISP, you must understand the differences among the data types and how to use them.

## Integers

Integers are whole numbers that do not contain a decimal point. AutoLISP integers are 32-bit signed numbers with values ranging from +2,147,483,647 to -2,147,483,648. (Note, however, that the `getint` function only accepts 16-bit numbers ranging from +32767 to -32768.) When you explicitly use an integer in an AutoLISP expression, that value is known as a constant. Numbers such as 2, -56, and 1,200,196 are valid AutoLISP integers.

If you enter a number that is greater than the maximum integer allowed (resulting in integer overflow), AutoLISP converts the integer to a real number. However, if you perform an arithmetic operation on two valid integers, and the result is greater than the maximum allowable integer, the resulting number will be invalid. The following examples illustrate how AutoLISP handles integer overflow.

The largest positive integer value retains its specified value:

```
_$ 2147483647  
2147483647
```

If you enter an integer that is greater than the largest allowable value, AutoLISP returns the value as a real:

```
_$ 2147483648  
2.14748e+009
```

An arithmetic operation involving two valid integers, but resulting in integer overflow, produces an invalid result:

```
_$ (+ 2147483646 3)  
-2147483647
```

In this example the result is clearly invalid, as the addition of two positive numbers results in a negative number. But note how the following operation produces a valid result:

```
_$ (+ 2147483648 2)  
2.14748e+009
```

In this instance, AutoLISP converts 2147483648 to a valid real before adding 2 to the number. The result is a valid real.

The largest negative integer value retains its specified value:

```
_$ -2147483647  
-2147483647
```

If you enter a negative integer larger than the greatest allowable negative value, AutoLISP returns the value as a real:

```
_$ -2147483648  
-2.14748e+009
```

The following operation concludes successfully, because AutoLISP first converts the overflow negative integer to a valid real:

```
_$ (- -2147483648 1)  
-2.14748e+009
```

## Reals

A real is a number containing a decimal point. Numbers between -1 and 1 must contain a leading zero. Real numbers are stored in double-precision floating-point format, providing at least 14 significant digits of precision. Note that VLISP does not show you all the significant digits.

Reals can be expressed in scientific notation, which has an optional *e* or *E* followed by the exponent of the number (for example, 0.0000041 is the same as 4.1e-6). Numbers such as 3.1, 0.23, -56.123, and 21,000,000.0 are valid AutoLISP reals.

## Strings

A string is a group of characters surrounded by quotation marks. Within quoted strings the backslash (\) character allows control characters (or escape codes) to be included. When you explicitly use a quoted string in an AutoLISP expression, that value is known as a literal string or a string constant.

Examples of valid strings are “string 1” and “\nEnter first point:”.

## Lists

An AutoLISP list is a group of related values separated by spaces and enclosed in parentheses. Lists provide an efficient method of storing numerous related values. AutoCAD expresses 3D points as a list of three real numbers.

Examples of lists are (1.0 1.0 0.0), (“this” “that” “the other”), and (1 “ONE”).

## Selection Sets

Selection sets are groups of one or more objects (entities). You can interactively add objects to, or remove objects from, selection sets with AutoLISP routines.

The following example uses the `ssget` function to return a selection set containing all the objects in a drawing.

```
_$ (ssget "X")  
<Selection set: 1>
```

## Entity Names

An entity name is a numeric label assigned to objects in a drawing. It is actually a pointer into a file maintained by AutoCAD, and can be used to find the object's database record and its vectors (if they are displayed). This label can be referenced by AutoLISP functions to allow selection of objects for processing in various ways. Internally, AutoCAD refers to objects as entities.

The following example uses the `entlast` function to get the name of the last object entered into the drawing.

```
_$ (entlast)  
<Entity name: 27f0540>
```

Entity names assigned to objects in a drawing are only in effect during the current editing session. The next time you open the drawing, AutoCAD assigns new entity names to the objects. You can use an object's handle to refer to it from one editing session to another; see "Entity Handles and Their Uses" on page 307 for information on using handles.

## VLA-objects

Objects in a drawing may be represented as Visual LISP ActiveX (VLA) objects, a data type introduced with Visual LISP. When working with ActiveX functions, you must refer to VLA-objects, not the `ename` pointer returned by functions such as `entlast`. For information on working with ActiveX objects, see "Using ActiveX Objects with AutoLISP" on page 162.

## File Descriptors

A file descriptor is a pointer to a file opened by the AutoLISP `open` function. The `open` function returns this pointer as an alphanumeric label. You supply

the file descriptor as an argument to other AutoLISP functions that read or write to the file.

The following example opens the *myinfo.dat* file for reading. The `open` function returns the file descriptor:

```
_$ (setq file1 (open "c:\\myinfo.dat" "r"))  
#<file "c:\\myinfo.dat">
```

In this example, the file descriptor is stored in the `file1` variable.

Files remain open until you explicitly close them in your AutoLISP program. The `close` function closes a file. The following code closes the file whose file descriptor is stored in the `file1` variable:

```
_$ (close file1)  
nil
```

## Symbols and Variables

AutoLISP uses symbols to refer to data. Symbol names are not case-sensitive and may consist of any sequence of alphanumeric and notation characters except the following:

### Characters restricted from symbol names

(	(Open Parenthesis)
---	--------------------

)	(Close Parenthesis)
---	---------------------

.	(Period)
---	----------

'	(Apostrophe)
---	--------------

"	(Quote Symbol)
---	----------------

;	(Semicolon)
---	-------------

A symbol name cannot consist of only numeric characters.

Technically, AutoLISP applications consist of either symbols or constant values, such as strings, reals, and integers. For the sake of clarity, this guide uses the term *symbol* to refer to a symbol name that stores static data, such as built-in and user-defined functions. The term *variable* is used to refer to a symbol name that stores program data. The following example uses the `setq`

function to assign the string value "this is a string" to the str1 variable:

```
_$ (setq str1 "this is a string")  
"this is a string"
```

Be kind to yourself and to others who need to read your code. Choose meaningful names for your program symbols and variables.

## Protected Symbols

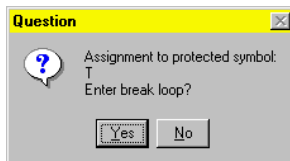
You may be warned if you attempt to change the value of some symbols used by the AutoLISP language. These symbols are known as protected symbols, and include items such as arithmetic operators (for example, +, -) and the values T and nil. You can use the Visual LISP Symbol Service feature to determine if a symbol is protected.

When you first start AutoCAD, protected symbols receive no special protection. If you set a protected symbol at the AutoCAD Command prompt, you receive no indication that the symbol has any special status. However, once you start Visual LISP, this changes. From the moment you start Visual LISP until the end of your AutoCAD session, AutoLISP intercepts any attempt to modify a protected symbol. Processing of protected symbols depends on the status of a Visual LISP environment option. You can specify one of the following options:

- **Transparent** Protected symbols are treated like any other symbol.
- **Print message** AutoLISP issues a warning message when you modify a protected symbol but carries out the modification. For example, the following demonstrates what happens when you modify the symbol T:

```
Command: (setq t "look out")  
; *U* WARNING: assignment to protected symbol: T <- "look out"  
"look out"
```

- **Prompt to enter break loop** This is the default option, resulting in AutoLISP displaying the following message box when you attempt to modify a protected symbol:



If you choose No, the symbol's value is modified, and processing continues normally. If you choose Yes, processing is interrupted, and you enter a Visual LISP break loop. Control switches to the Visual LISP Console window. To set the symbol and continue processing, press the Continue



button on the Visual LISP toolbar; to abort modification, press Reset.

- **Error** This option prohibits modification of protected symbols. Any attempt to modify a protected symbol results in an error message.

To specify how AutoLISP responds to attempts to modify protected symbols, choose Tools ► Environment Options ► General Options from the Visual LISP menu.

## AutoLISP Program Files

Although you can enter AutoLISP code in the Visual LISP Console window or at the AutoCAD Command prompt, testing and debugging a series of instructions are considerably easier when you save AutoLISP code in a file rather than re-entering it each time you make a refinement. AutoLISP source code is usually stored in ASCII text files with an *.lsp* extension. However, you can load AutoLISP code from any ASCII text file.

To create a new source file in VLISP, choose New File from the VLISP File menu.

To edit existing AutoLISP source code in VLISP, choose Open File from the VLISP File menu, and use the Open File dialog box to select your file. VLISP loads this file into its text editor and displays the contents in a new editor window.

## Formatting AutoLISP Code

The extensive use of parentheses in AutoLISP code can make it difficult to read. The traditional technique for combatting this confusion is indentation. The more deeply nested a line of code is, the farther to the right you position the line.

If you use the VLISP text editor to enter your code, VLISP automatically formats the code as you enter it. VLISP also has features to reformat a selection or an entire file. This improves the appearance of your code, making it more readable. For information on using these features, see “Formatting Code with Visual LISP” on page 50.

## Spaces in AutoLISP Code

In AutoLISP, multiple spaces between variable names, constants, and function names are equivalent to a single space. The end of a line is also treated as a single space.

The following two expressions produce the same result:

```
(setq test1 123 test2 456)
(setq
  test1 123
  test2 456
)
```

## Comments

It is good practice to include comments in AutoLISP program files. Comments are useful to both the programmer and future users who may need to revise a program to suit their needs. Use comments to do the following:

- Give a title, authorship, and creation date
- Provide instructions on using a routine
- Make explanatory notes throughout the body of a routine
- Make notes to yourself during debugging

Comments begin with one or more semicolons (;) and continue through the end of the line.

```
; This entire line is a comment
(setq area (* pi r r)) ; Compute area of circle
```

Any text within ; | ... |; is ignored. Therefore, comments can be included within a line of code or extend for multiple lines. This type of comment is known as an in-line comment.

```
(setq tmode ; |some note here|; (getvar "tilemode"))
```

The following example shows a comment that continues for multiple lines:

```
(setvar "orthomode" 1) ; |comment starts here
and continues to this line,
but ends way down here|; (princ "\nORTHOMODE set On.")
```

It is recommended that you use comments liberally when writing AutoLISP programs. The tutorial files provided with VLISP contain good examples of commenting style. If you've installed the AutoCAD samples files, you'll find the VLISP tutorial code in the *Tutorial\VisualLISP* directory.

## Visual LISP Comment Styles

The VLISP code formatter recognizes five types of comments and positions each comment according to its type. Refer to "Applying Visual LISP Comment Styles" on page 58 for a description of each comment style.

Regardless of your commenting style, it is more important that comments be present rather than they obey any particular layout rules.

## Color Coding

VLISP provides an additional solution to make AutoLISP text easier to read: color coding. VLISP looks at each word of text and tries to determine what type of AutoLISP language element the word represents (for example, a built-in function, a number, or a string). Every type of element is assigned its own color, so you can easily distinguish among them when viewing the code. See “Understanding Visual LISP Color Coding” on page 42 for more information on the VLISP color coding feature.

Keep in mind that color coding is a VLISP text editor feature, and it is possible that someone who does not have access to VLISP may need to read your code some day. For this reason, you should still use indentation and alignment to enhance your program’s readability.

## AutoLISP Variables

An AutoLISP variable assumes the data type of the value assigned to it. Until they are assigned new values, variables retain their original values. You use the AutoLISP `setq` function to assign values to variables.

```
(setq variable_name1 value1 [variable_name2 value2 ...])
```

The `setq` function assigns the specified value to the variable name given. It returns the value as its function result. If you issue `setq` at the Visual LISP Console prompt, the result is displayed in the Console window:

```
_$ (setq val 3 abc 3.875)
3.875
_$ (setq layr "EXTERIOR-WALLS")
"EXTERIOR-WALLS"
_$
```

## Displaying the Value of a Variable

To display the current value of a variable when working in Visual LISP, just enter the variable name at the Console prompt as follows:

```
_$ abc
3.875
```

To display the value of a variable from the AutoCAD Command prompt, you must precede the variable name with an exclamation point (!). For example:

```
Command: !abc  
3.875
```

## Nil Variables

An AutoLISP variable that has not been assigned a value is said to be `nil`. This is different from blank, which is considered a character string, and different from 0, which is a number. So, in addition to checking a variable for its current value, you can test to determine if the variable has been assigned a value.

Each variable consumes a small amount of memory, so it is good programming practice to reuse variable names or set variables to `nil` when their values are no longer needed. Setting a variable to `nil` releases the memory used to store that variable's value. If you no longer need the `val` variable, you can release its value from memory with the following expression:

```
_$ (setq val nil)  
nil
```

Another efficient programming practice is to use local variables whenever possible. See “Local Variables in Functions” on page 251 on this topic.

## Predefined Variables

The following predefined variables are commonly used in AutoLISP applications:

PAUSE	Defined as a string consisting of a double backslash (\\) character. This variable is used with the <code>command</code> function to pause for user input.
PI	Defined as the constant $\pi$ (pi). It evaluates to approximately 3.14159.
T	Defined as the constant T. This is used as a non- <code>nil</code> value.

---

**NOTE** You can change the value of these variables with the `setq` function. However, other applications might rely on their values being consistent; therefore, it is recommended that you do not modify these variables. Visual LISP, by default, protects these variables from redefinition. You can override this protection through the VLISP Symbol Service feature or by setting a VLISP environment option.

---

## Number Handling

AutoLISP provides functions for working with integers and real numbers. In addition to performing complex mathematical computations in applications, you can use the number-handling functions to help you in your daily use of AutoCAD. If you are drawing a steel connection detail that uses a 2.5" bolt that is 0.5" in diameter, how many threads are there if the bolt has 13 threads per inch? Use the `*` (multiplication) function at the Console prompt, as in the following example:

```
_$ (* 2.5 13)  
32.5
```

The arithmetic functions that have a number argument (as opposed to `num` or `angle`, for example) return different values if you provide integers or reals as arguments. If all arguments are integers, the value returned is an integer. However, if one or all the arguments are reals, the value returned is a real. To ensure your application passes real values, be certain at least one argument is a real.

```
_$ (/ 12 5)  
2  
_$ (/ 12.0 5)  
2.4
```

A complete list of number-handling functions is in appendix A, "AutoLISP Function Synopsis," under the heading "Arithmetic Functions." These functions are described in the *AutoLISP Reference*.

# String Handling

AutoLISP provides functions for working with string values. For example, the `strcase` function returns the conversion of all alphabetic characters in a string to uppercase or lowercase. It accepts two arguments: a string and an optional argument that specifies the case in which the characters are returned. If the optional second argument is omitted, it evaluates to `nil` and `strcase` returns the characters converted to uppercase.

```
_$ (strcase "This is a TEST.")  
"THIS IS A TEST."
```

If you provide a second argument of `T`, the characters are returned as lowercase. AutoLISP provides the predefined variable `T` to use in similar situations where a non-`nil` value is used as a type of true/false toggle.

```
_$ (strcase "This is a TEST." T)  
"this is a test."
```

The `strcat` function combines multiple strings into a single string value. This is useful for placing a variable string within a constant string. The following code sets a variable to a string value and then uses `strcat` to insert that string into the middle of another string.

```
_$ (setq str "BIG") (setq bigstr (strcat "This is a " str " test."))  
"This is a BIG test."
```

If the variable `bigstr` is set to the preceding string value, you can use the `strlen` function to find out the number of characters (including spaces) in that string.

```
_$ (strlen bigstr)  
19
```

The `substr` function returns a substring of a string. It has two required arguments and one optional argument. The first required argument is the string. The second argument is a positive integer that specifies the first character of the string you want to include in the substring. If the third argument is provided, it specifies the number of characters to include in the substring. If the third argument is not provided, `substr` returns all characters including and following the specified start character.

As an example, you can use the `substr` function to strip off the three-letter extension from a file name (note that you can actually use the

vl -filenam-base function to do this). First, set a variable to a file name.

```
_$ (setq filnam "bigfile.txt")
"bigfile.txt"
```

You need to get a string that contains all characters except the last four (the period and the three-letter extension). Use `strlen` to get the length of the string and subtract 4 from that value. Then use `substr` to specify the first character of the substring and its length.

```
_$ (setq newlen (- (strlen filnam) 4))
7
_$ (substr filnam 1 newlen)
"bigfile"
```

If your application has no need for the value of `newlen`, you can combine these two lines of code into one.

```
_$ (substr filnam 1 (- (strlen filnam) 4))
"bigfile"
```

Additional string-handling functions are listed in appendix A, “AutoLISP Function Synopsis,” under the heading “String-Handling Functions.” These functions are described in the *AutoLISP Reference*.

AutoLISP also provides a number of functions that convert string values into numeric values and numeric values into string values. These functions are discussed in “Conversions” on page 279.

## Basic Output Functions

AutoLISP includes functions for controlling the AutoCAD display, including both text and graphics windows. Some functions also display information in the Visual LISP Console window. The major text display functions are:

- `prinl`
- `princ`
- `print`
- `prompt`

These functions are discussed in the following sections. The remaining display functions are covered in chapter 9, “Using AutoLISP to Communicate with AutoCAD”, beginning with the “Display Control” topic.

## Displaying Messages

When entered from VLISP, the `prompt` function displays a message (a string) in the AutoCAD Command window and returns `nil` to the VLISP Console window. The `princ`, `prin1`, and `print` functions all display an expression (not necessarily a string) in the AutoCAD Command window and return the expression to the VLISP Console window. Optionally, these functions can send output to a file. The differences are as follows:

- `princ` displays strings without the enclosing quotation marks.
- `prin1` displays strings enclosed in quotation marks.
- `print` displays strings enclosed in quotation marks but places a blank line before the expression and a space afterward.

The following examples demonstrate the differences between the four basic output functions and how they handle the same string of text. If you enter the examples from VLISP, the text following *prints* is what you see at the AutoCAD Command prompt; text following *returns* appears within the VLISP Console window or within an application. See the following section for an explanation of the control characters used in the example.

```
(setq str "The \"allowable\" tolerance is ± 1/4")  
(prompt str) prints      The "allowable" tolerance is ± 1/4"  
               and returns nil  
(princ str)   prints      The "allowable" tolerance is ± 1/4"  
               and returns "The \"allowable\" tolerance is ± 1/4"  
(prin1 str)   prints      "The \"allowable\" tolerance is ± 1/4"  
               and returns "The \"allowable\" tolerance is ± 1/4"  
(print str)   prints      <blank line>  
               and returns "The \"allowable\" tolerance is ± 1/4" <space>  
                        "The \"allowable\" tolerance is ± 1/4"
```

Note that the `writeln` and `writeln` functions can also display output to a Command window. Refer to the *AutoLISP Reference* for information on these functions.

### Exiting Quietly

If you invoke the `princ` function without passing an expression to it, it displays nothing and has no value to return. So if you write an AutoLISP expression that ends with a call to `princ` without any arguments, the ending `nil` is suppressed (because it has nothing to return). This practice is called exiting quietly.



## Control Characters in Strings

Within quoted strings, the backslash (\) character allows control characters (or escape codes) to be included. The following table shows the currently recognized control characters:

AutoLISP control characters	
Code	Description
\\	\ character
\"	" character
\e	Escape character
\n	Newline character
\r	Return character
\t	Tab character
\nnn	Character whose octal code is <i>nnn</i>

The `prompt` and `princ` functions expand the control characters in a string and display the expanded string in the AutoCAD Command window.

If you need to use the backslash character (\) or quotation mark (") within a quoted string, it must be preceded by the backslash character (\). For example, if you enter

```
_$ (princ "The \"filename\" is: D:\\ACAD\\TEST.TXT. ")
```

the following displays in the AutoCAD Command window:

```
The "filename" is: D:\ACAD\TEST.TXT
```

You will also see this output in the VLISP Console window, along with the return value from the `princ` function (which is your original input, with the unexpanded control characters).

To force a line break at a specific location in a string, use the newline character (\n).

```
_$ (prompt "An example of the \nnewline character. ")
```

```
An example of the  
newline character.
```

You can also use the `terpri` function to cause a line break.

The return character (`\r`) returns to the beginning of the current line. This is useful for displaying incremental information (for example, a counter showing the number of objects processed during a loop).

The Tab character (`\t`) can be used in strings to indent or to provide alignment with other tabbed text strings. In this example, note the use of the `princ` function to suppress the ending `nil`.

```
_$ (prompt "\nName\tOffice\n-----\t-----")  
(_> \nSue\t101\nJoe\t102\nSam\t103\n") (princ)
```

Name	Office
-----	-----
Sue	101
Joe	102
Sam	103

## Wild-Card Matching

The `wcmatch` function enables applications to compare a string to a wild-card pattern. You can use this facility when you build a selection set (in conjunction with `ssget`) and when you retrieve extended entity data by application name (in conjunction with `entget`).

The `wcmatch` function compares a single string to a pattern. The function returns `T` if the string matches the pattern, and `nil` if it does not. The wild-card patterns are similar to the regular expressions used by many system and application programs. In the pattern, alphabetic characters and numerals are treated literally; brackets can be used to specify optional characters or a range of letters or digits; a question mark (?) matches a single character; an asterisk (\*) matches a sequence of characters; and, certain other special characters have special meanings within the pattern. When you use the \* character at the beginning and end of the search pattern, you can locate the desired portion anywhere in the string. For more information, see “`wcmatch`” in the *AutoLISP Reference*.

In the following examples, a string variable called `matchme` has been declared and initialized:

```
_ $ (setq matchme "this is a string - test1 test2 the end")
"this is a string - test1 test2 the end"
```

The following code checks whether or not `matchme` begins with the four characters "thi s":

```
_ $ (wcmatch matchme "this*")
T
```

The following code illustrates the use of brackets in the pattern. In this case, `wcmatch` returns `T` if `matchme` contains "test4", "test5", "test6" (4-6), or "test9" (note the use of the `*` character):

```
_ $ (wcmatch matchme "**test[4-69]*")
nil
```

In this case, `wcmatch` returns `nil` because `matchme` does not contain any of the strings indicated by the pattern.

However,

```
_ $ (wcmatch matchme "**test[4-61]*")
T
```

because the string contains "test1".

The pattern string can specify multiple patterns, separated by commas. The following code returns `T` if `matchme` equals "ABC", or if it begins with "XYZ", or if it ends with "end".

```
_ $ (wcmatch matchme "ABC,XYZ*,*end")
T
```

# Equality and Conditional

AutoLISP includes functions that provide equality verification as well as conditional branching and looping. The equality and conditional functions are listed in appendix A, “AutoLISP Function Synopsis,” under the heading “Equality and Conditional Functions.” These functions are described in the *AutoLISP Reference*.

When writing code that checks string and symbol table names, keep in mind that AutoLISP automatically converts symbol table names to upper case in some instances. When testing symbol names for equality, you need to make the comparison insensitive to the case of the names. Use the `strcase` function to convert strings to the same case before testing them for equality.

## List Handling

AutoLISP provides functions for working with lists. This section provides examples of the `append`, `assoc`, `car`, `cons`, `list`, `nth`, and `subst` functions. A summary of all list-handling functions is in appendix A, “AutoLISP Function Synopsis,” under the heading “List Manipulation Functions.” Each list-handling function is described in the *AutoLISP Reference*.

Lists provide an efficient and powerful method of storing numerous related values. After all, LISP is so-named because it is the LIST Processing language. Once you understand the power of lists, you’ll find that you can create more powerful and flexible applications.

Several AutoLISP functions provide a basis for programming two-dimensional and three-dimensional graphics applications. These functions return point values in the form of a list.

The `list` function provides a simple method of grouping related items. These items do not need to be of similar data types. The following code groups three related items as a list:

```
_$ (setq lst1 (list 1.0 "One" 1))  
(1.0 "One" 1)
```

You can retrieve a specific item from the list in the `lst1` variable with the `nth` function. This function accepts two arguments. The first argument is an integer that specifies which item to return. A 0 specifies the first item in a list,

1 specifies the second item, and so on. The second argument is the list itself. The following code returns the second item in `lst1`.

```
_$ (nth 1 lst1)
"One"
```

The `cdr` function returns all elements, except the first, from a list. For example:

```
_$ (cdr lst1)
("One" 1)
```

The `car` function provides another way to extract items from a list. For more examples using `car` and `cdr`, and combinations of the two, see “Point Lists” on page 242.

Three functions let you modify an existing list. The `append` function returns a list with new items added to the end of it, and the `cons` function returns a list with new items added to the beginning of the list. The `subst` function returns a list with a new item substituted for every occurrence of an old item. These functions do not modify the original list, they return a modified list. To modify the original list, you must explicitly replace the old list with the new list.

The `append` function takes any number of lists and runs them together as one list. Therefore, all arguments to this function must be lists. The following code adds another “One” to the list `lst1`. Note the use of the `quote` (or `'`) function as an easy way to make the string “One” into a list.

```
_$ (setq lst2 (append lst1 '("One")))
(1.0 "One" 1 "One")
```

The `cons` function combines a single element with a list. You can add another string “One” to the beginning of this new list, `lst2`, with the `cons` function.

```
_$ (setq lst3 (cons "One" lst2))
("One" 1.0 "One" 1 "One")
```

You can substitute all occurrences of an item in a list with a new item with the `subst` function. The following code replaces all strings “One” with the string “one”.

```
_$ (setq lst4 (subst "one" "One" lst3))
("one" 1.0 "one" 1 "one")
```

## Point Lists

AutoLISP observes the following conventions for handling graphics coordinates. Points are expressed as *lists* of two or three numbers surrounded by parentheses.

2D points	Expressed as lists of two real numbers ( <i>X</i> and <i>Y</i> , respectively), as in  (3. 4 7. 52)
3D points	Expressed as lists of three real numbers ( <i>X</i> , <i>Y</i> , and <i>Z</i> , respectively), as in  (3. 4 7. 52 1. 0)

You can use the `list` function to form point lists, as shown in the following examples:

```
_$ (list 3.875 1.23)
(3.875 1.23)
_$ (list 88.0 14.77 3.14)
(88.0 14.77 3.14)
```

To assign particular coordinates to a point variable, you can use one of the following expressions:

```
_$ (setq pt1 (list 3.875 1.23))
(3.875 1.23)

_$ (setq pt2 (list 88.0 14.77 3.14))
(88.0 14.77 3.14)

_$ (setq abc 3.45)
3.45

_$ (setq pt3 (list abc 1.23))
(3.45 1.23)
```

The latter uses the value of variable `abc` as the *X* component of the point.

If all members of a list are constant values, you can use the `quote` function to explicitly define the list, rather than the `list` function. The `quote` function returns an expression without evaluation, as follows:

```
_$ (setq pt1 (quote (4.5 7.5)))
(4.5 7.5)
```

The single quotation mark ( ' ) can be used as shorthand for the quote function. The following code produces the same result as the preceding code.

```
_ $ (setq pt1 ' (4.5 7.5))  
(4.5 7.5)
```

You can refer to *X*, *Y*, and *Z* components of a point individually, using three additional built-in functions called `car`, `cadr`, and `caddr`. The following examples show how to extract the *X*, *Y*, and *Z* coordinates from a 3D point list. The `pt` variable is set to the point (1.5 3.2 2.0):

```
_ $ (setq pt ' (1.5 3.2 2.0))  
(1.5 3.2 2.0)
```

The `car` function returns the first member of a list. In this example it returns the *X* value of point `pt` to the `x_val` variable.

```
_ $ (setq x_val (car pt))  
1.5
```

The `cadr` function returns the second member of a list. In this example it returns the *Y* value of the `pt` point to the `y_val` variable.

```
_ $ (setq y_val (cadr pt))  
3.2
```

The `caddr` function returns the third member of a list. In this example it returns the *Z* value of point `pt` to the variable `z_val`.

```
_ $ (setq z_val (caddr pt))  
2.0
```

You can use the following code to define the lower-left and upper-right (`pt1` and `pt2`) corners of a rectangle, as follows:

```
_ $ (setq pt1 ' (1.0 2.0) pt2 ' (3.0 4.0))  
(3.0 4.0)
```

You can use the `car` and `cadr` functions to set the `pt3` variable to the upper-left corner of the rectangle, by extracting the *X* component of `pt1` and the *Y* component of `pt2`, as follows:

```
_ $ (setq pt3 (list (car pt1) (cadr pt2)))  
(1.0 4.0)
```

The preceding expression sets `pt3` equal to point (1.0,4.0).

AutoLISP supports concatenations of `car` and `cdr` up to four levels deep. The following are valid functions:

caaaaar	cadaaar	cdaaar	cddaar
caaadr	cadadr	cdaadr	cddadr
caaar	cadar	cdaar	cddar
caadar	caddar	cdadar	cdddar
caaddr	caddr	cdaddr	cdddr
caadr	caddr	cdadr	cdddr
caar	cadr	cdar	cddr

These concatenations are the equivalent of nested calls to `car` and `cdr`. Each *a* represents a call to `car`, and each *d* represents a call to `cdr`. For example:

<code>(caar x)</code>	<i>is equivalent to</i>	<code>(car (car x))</code>
<code>(cdar x)</code>	<i>is equivalent to</i>	<code>(cdr (car x))</code>
<code>(cadar x)</code>	<i>is equivalent to</i>	<code>(car (cdr (car x)))</code>
<code>(cadr x)</code>	<i>is equivalent to</i>	<code>(car (cdr x))</code>
<code>(cddr x)</code>	<i>is equivalent to</i>	<code>(cdr (cdr x))</code>
<code>(caddr x)</code>	<i>is equivalent to</i>	<code>(car (cdr (cdr x)))</code>

## Dotted Pairs

Another way AutoCAD uses lists to organize data is with a special type of list called a dotted pair. This list must always contain two members. When representing a dotted pair, AutoLISP separates the members of the list with a period (.). Most list-handling functions will not accept a dotted pair as an argument, so you should be sure you are passing the right kind of list to a function.

Dotted pairs are an example of an "improper list." An improper list is one in which the last `cdr` is not `nil`. In addition to adding an item to the beginning of a list, the `cons` function can create a dotted pair. If the second argument to the `cons` function is anything other than another list or `nil`, it creates a dotted pair.

```
_$ (setq sublist (cons 'lyr "WALLS"))
(LYR . "WALLS")
```

The `car`, `cdr`, and `assoc` functions are useful for handling dotted pairs. The following code creates an association list, which is a list of lists, and is the method AutoCAD uses to maintain entity definition data. (Entity definition



data is discussed in chapter 10, “Using AutoLISP to Manipulate AutoCAD Objects.”) The following code creates an association list of dotted pairs:

```
_$ (setq wallinfo (list sublist(cons 'len 240.0) (cons 'hgt 96.0)))  
( (LYR . "WALLS") (LEN . 240.0) (HGT . 96.0) )
```

The `assoc` function returns a specified list from within an association list regardless of the specified list’s location within the association list. The `assoc` function searches for a specified key element in the lists, as follows:

```
_$ (assoc 'len wallinfo)  
(LEN . 240.0)  
_$ (cdr (assoc 'lyr wallinfo))  
"WALLS"  
_$ (nth 1 wallinfo)  
(LEN . 240.0)  
_$ (car (nth 1 wallinfo))  
LEN
```

## Symbol and Function Handling

AutoLISP provides a number of functions for handling symbols and variables. The symbol-handling functions are listed in appendix A, “AutoLISP Function Synopsis,” under the heading “Symbol-Handling Functions.” Each symbol-handling function is described in the *AutoLISP Reference*.

AutoLISP provides functions for handling one or more groups of functions. This section provides examples of the `defun` function. The remaining function-handling functions are listed in appendix A, “AutoLISP Function Synopsis,” under the heading “Function-Handling Functions.” The functions are described in the *AutoLISP Reference*.

### Using `defun` to Define a Function

With AutoLISP you can define your own functions. Once defined, you can use these functions at the AutoCAD Command prompt, the Visual LISP Console prompt, or within other AutoLISP expressions, just as you use the standard functions. You can also create your own AutoCAD commands, because commands are just a special type of function.

The `defun` function (see “`defun`” in the *AutoLISP Reference*) combines a group of expressions into a function or command. This function requires at least three arguments, the first of which is the name of the function (symbol name) to define. The second argument is the argument list (a list of argu-

ments and local variables used by the function). The argument list can be `nil` or an empty list `()`. Argument lists are discussed in greater detail in “Functions with Arguments” on page 252. If local variables are provided, they are separated from the arguments by a slash `(/)`. Local variables are discussed in “Local Variables in Functions” on page 251. Following these arguments are the expressions that make up the function; there must be at least one expression in a function definition.

```
(defun symbol_name ( args / local_variables )
  expressions
)
```

The following code defines a simple function that accepts no arguments and displays “bye” in the AutoCAD Command window. Note that the argument list is defined as an empty list `()`:

```
_$ (defun DONE () (prompt "\nbye! "))
DONE
```

Now that the `DONE` function is defined, you can use it as you would any other function. For example, the following code prints a message, then says “bye” in the AutoCAD Command window:

```
_$ (prompt "The value is 127.") (DONE) (princ)
The value is 127
bye!
```

Note how the previous example invokes the `princ` function without any arguments. This suppresses an ending `nil` and achieves a quiet exit.

Functions that accept no arguments may seem useless. However, you might use this type of function to query the state of certain system variables or conditions and to return a value that indicates those values.

AutoCAD can automatically load your functions each time you start a new AutoCAD session or open a new AutoCAD drawing file; see “Automatic Loading and Execution of AutoLISP Routines” in the AutoCAD *Customization Guide* for further information on automatic loading.

Any code in an AutoLISP program file that is not part of a `defun` statement is executed when that file is loaded. You can use this to set up certain parameters or to perform any other initialization procedures in addition to displaying textual information, such as how to invoke the loaded function.

## Compatibility of `defun` with Previous Versions of AutoCAD

The internal implementation of `defun` changed in AutoCAD 2000. This change will be transparent to the great majority of AutoLISP users upgrading from earlier versions of AutoCAD. The change only affects AutoLISP code

that manipulated `defun` definitions as a list structure, such as by appending one function to another, as in the following code:

```
(append s::startup (cdr mystartup))
```

For situations like this, you can use `defun-q` to define your functions. An attempt to use a `defun` function as a list results in an error. The following example illustrates the error:

```
_$ (defun foo (x) 4)
foo
_$ (append foo ' (3 4))
; error: Invalid attempt to access a compiled function definition.
You may want to define it using defun-q: #<SUBR @024bda3c F00>
```

The error message alerts you to the possibility of using `defun-q` instead of `defun`.

The `defun-q` function is provided strictly for backward compatibility with previous versions of AutoLISP and should not be used for other purposes. For more information on using `defun-q`, and the related `defun-q-list-set` and `defun-q-list-ref` functions, see the *AutoLISP Reference*.

## C:XXX Functions

If an AutoLISP function is defined with a name of the form `C:XXX`, it can be issued at the AutoCAD Command prompt in the same manner as a built-in AutoCAD command. This is true regardless of whether you define and load the function in VLISP or at the AutoCAD Command prompt. You can use this feature to add new commands to AutoCAD or to redefine existing commands.

To use functions as AutoCAD commands, be sure they adhere to the following rules:

- The function name must use the form `C:XXX` (upper- or lowercase characters). The `C:` portion of the name must always be present; the `XXX` portion is a command name of your choice. `C:XXX` functions can be used to override built-in AutoCAD commands. (See “Redefining AutoCAD Commands” on page 249.)
- The function must be defined with no arguments. However, local variables are permitted and it is a good programming practice to use them.

A function defined in this manner can be issued transparently from within any prompt of any built-in AutoCAD command, provided the function issued transparently does not call the `command` function. (This is the AutoLISP function you use to issue AutoCAD commands; see the entry on “`command`” in the *AutoLISP Reference*.) When issuing a `C:XXX` defined command transpar-

ently, you must precede the *XXX* portion with a single quotation mark ( ' ).

You can issue a built-in command transparently while a C: *XXX* command is active by preceding it with a single quotation mark ( ' ), as you would with all commands that are issued transparently. However, you cannot issue a C: *XXX* command transparently while a C: *XXX* command is active.

---

**NOTE** When calling a function defined as a command from the code of another AutoLISP function, you must use the whole name, including the parentheses; for example, (C: HELLO). You also must use the whole name and the parentheses when you invoke the function from the VLISP Console prompt.

---

## Adding Commands

Using the C: *XXX* feature, you can define a command that displays a simple message.

```
_$ (defun C:HELLO () (princ "Hello world. \n") (princ))
C:HELLO
```

HELLO is now defined as a command, in addition to being an AutoLISP function. This means you can issue the command from the AutoCAD Command prompt.

```
Command: hello
Hello world.
```

This new command can be issued transparently because it does not call the command function itself. At the AutoCAD Command prompt, you could do the following:

```
Command: line
From point: ' hello
Hello world.
From point:
```

Remember: to run this function from the VLISP Console window, you need to issue the function call with the parentheses because VLISP does not recognize AutoCAD commands.

```
_$ (c:hello)
Hello world.
```

If you follow your function definition with a call to the `setfunhelp` function, you can associate a Help file and topic with a user-defined command. When help is requested during execution of the user-defined command, the topic specified by `setfunhelp` displays. See the *AutoLISP Reference* for more information on using `setfunhelp`.

You cannot usually use an AutoLISP statement to respond to prompts from an AutoLISP-implemented command. However, if your AutoLISP routine makes use of the `initget` function, you can use arbitrary keyboard input with certain functions. This allows an AutoLISP-implemented command to accept an AutoLISP statement as a response. Also, the values returned by a DIESEL expression can perform some evaluation of the current drawing and return these values to AutoLISP. See “Keyword Options” on page 272 for more information on using `initget`, and refer to the AutoCAD *Customization Guide* for information on the DIESEL string expression language.

## Redefining AutoCAD Commands

Using AutoLISP, external commands, and the alias feature, you can define your own AutoCAD commands. You can use the `UNDEFINE` command to redefine a built-in AutoCAD command with a user-defined command of the same name. To restore the built-in definition of a command, use the `REDEFINE` command. The `UNDEFINE` command is in effect for the current editing session only.

You can always activate an undefined command by specifying its true name, which is the command name prefixed by a period. For example, if you undefine `QUIT`, you can still access the command by entering `.quit` at the AutoCAD Command prompt. This is also the syntax that should be used within the AutoLISP `command` function.

Consider the following example. Whenever you use the `LINE` command, you want AutoCAD to remind you about using the `PLINE` command. You can define the AutoLISP function `C:LINE` to substitute for the normal `LINE` command as follows:

```
_$ (defun C:LINE ()  
  (> (princ "Shouldn't you be using PLINE?\n")  
  (> (command ".LINE") (princ) )  
  C:LINE
```

In this example, the function `C:LINE` is designed to issue its message and then to execute the normal `LINE` command (using its true name, `.LINE`). Before AutoCAD will use your new definition for the `LINE` command, you

must undefine the built-in LINE command. Enter the following to undefine the built-in LINE command:

```
_$ (command "undefine" "line")
```

Now, if you enter **line** at the AutoCAD Command prompt, AutoCAD uses the C: LINE AutoLISP function:

```
Command: line  
Shouldn't you be using PLINE?  
.LINE Specify first point: Specify first point:
```

The previous code example assumes the CMDECHO system variable is set to 1 (On). If CMDECHO is set to 0 (Off), AutoCAD does not echo prompts during a command function call. The following code uses the CMDECHO system variable to prevent the LINE command prompt from repeating:

```
_$ (defun C:LINE (/ cmdsave)  
  (> (setq cmdsave (getvar "cmdecho"))  
  (> (setvar "cmdecho" 0)  
  (> (princ "Shouldn't you be using PLINE?\n")  
  (> (command ".LINE")  
  (> (setvar "cmdecho" cmdsave)  
  (> (princ) )  
C:LINE
```

Now if you enter **l i n e** at the AutoCAD Command prompt, the following displays:

```
Shouldn't you be using PLINE?  
Specify first point:
```

You can use this feature in a drawing management system, for example. You can redefine the NEW, OPEN, and QUIT commands to write billing information to a log file before you terminate the editing session.

It is recommended that you protect your menus, scripts, and AutoLISP programs by using the period-prefixed forms of all commands. This ensures that your applications use the built-in command definitions rather than a redefined command.

See the “Command Search Procedure” topic in the AutoCAD *Customization Guide* for a description of the steps AutoCAD takes to evaluate command names.

## Local Variables in Functions

AutoLISP provides a method for defining a list of symbols (variables) that are available only to your function. These are known as local variables.

### Local Variables versus Global Variables

The use of local variables ensures that the variables in your functions are unaffected by the surrounding application and that your variables do not remain available after the calling function has completed its task.

Many user-defined functions are used as utility functions within larger applications. User-defined functions also typically contain a number of variables whose values and use are specific to that function.

The danger in using global variables, instead of local variables, is you may inadvertently modify them outside of the function they were declared in and intended for. This can lead to unpredictable behavior, and it can be very difficult to identify the source of this type of problem.

Another advantage of using local variables is that AutoCAD can recycle the memory space used by these variables, whereas global variables keep accumulating within AutoCAD memory space.

There are some legitimate uses for global variables, but these should be kept to a minimum. It is also a good practice to indicate that you intend a variable to be global. A common way of doing this is to add an opening and closing asterisk to the variable name, for example, `*default t-l ayer*`.

### Example Using Local Variables

The following example shows the use of local variables in a user-defined function (be certain there is at least one space between the slash and the local variables).

```
_$ (defun LOCAL ( / aaa bbb)
  (> (setq aaa "A" bbb "B")
  (> (princ (strcat "\naaa has the value " aaa ))
  (> (princ (strcat "\nbbb has the value " bbb))
  (> (princ) )
  LOCAL
```

Before you test the new function, assign variables `aaa` and `bbb` to values other than those used in the `LOCAL` function.

```
_$ (setq aaa 1 bbb 2)
2
```

You can verify that the variables `aaa` and `bbb` are actually set to those values.

```
_ $  aaa
1
_ $  bbb
2
```

Now test the `LOCAL` function.

```
_ $  (local)
aaa has the value A
bbb has the value B
```

You will notice the function used the values for `aaa` and `bbb` that are local to the function. You can verify that the current values for `aaa` and `bbb` are still set to their nonlocal values.

```
_ $  aaa
1
_ $  bbb
2
```

In addition to ensuring that variables are local to a particular function, this technique also ensures the memory used for those variables is available for other functions.

## Functions with Arguments

With AutoLISP you can define functions that accept arguments. Unlike many of the standard AutoLISP functions, user-defined functions cannot have optional arguments. When you call a user-defined function that accepts arguments, you must provide values for all the arguments.

The symbols to use as arguments are defined in the argument list before the local variables. Arguments are treated as a special type of local variable; argument variables are not available outside the function. You cannot define a function with multiple arguments of the same name.

The following code defines a function that accepts two string arguments, combines them with another string, and returns the resulting string.

```
_ $  (defun ARGTEST ( arg1 arg2 / ccc )
_ >  (setq ccc "Constant string")
_ >  (strcat ccc " , " arg1 " , " arg2 )
ARGTEST
```



The ARGTEST function returns the desired value because AutoLISP always returns the results of the last expression it evaluates. The last line in ARGTEST uses strcat to concatenate the strings, and the resulting value is returned. This is one example where you should not use the princ function to suppress the return value from your program.

This type of function can be used a number of times within an application to combine two variable strings with one constant string in a specific order. Because it returns a value, you can save the value to a variable for use later in the application.

```
_$ (setq newstr (ARGTEST "String 1" "String 2") )  
"Constant string, String 1, String 2"
```

The newstr variable is now set to the value of the three strings combined.

Note that the ccc variable was defined locally within the ARGTEST function. Once the function runs to completion, AutoCAD recycles the variable, recapturing the memory allocated to it. To prove this, check from the VLISP Console window to see if there is still a value assigned to ccc.

```
_$ ccc  
nil
```

## Special Forms

Certain AutoLISP functions are considered special forms because they evaluate arguments in a different manner than most AutoLISP function calls. A typical function evaluates all arguments passed to it before acting on those arguments. Special forms either do not evaluate all their arguments, or only evaluate some arguments under certain conditions.

The following AutoLISP functions are considered special forms:

- AND
- COMMAND
- COND
- DEFUN
- DEFUN-Q
- FOREACH
- FUNCTION
- IF
- LAMBDA
- OR
- PROGN
- QUOTE

- REPEAT
- SETQ
- TRACE
- UNTRACE
- VLAX-FOR
- WHILE

You can read about each of these functions in the *AutoLISP Reference*.

## Error Handling

The AutoLISP language provides several functions for error handling. You can use these functions to do the following:

- Provide information to users when an error occurs during the execution of a program.
- Restore the AutoCAD environment to a known state.
- Intercept errors and continue program execution.

The complete list of error-handling functions is in appendix A, “AutoLISP Function Synopsis,” under the heading “Error-Handling Functions.” Each error-handling function is described in the *AutoLISP Reference*.

If your program contains more than one error in the same expression, you cannot depend on the order in which AutoLISP detects the errors. For example, the `inters` function requires several arguments, each of which must be either a 2D or 3D point list. A call to `inters` like the following:

```
(inters 'a)
```

is an error on two counts: too few arguments and invalid argument type. You will receive either of the following error messages:

```
; *** ERROR: too few arguments
; *** ERROR: bad argument type: 2D/3D point
```

Your program should be designed to handle either error.

Note also that in AutoCAD 2000, AutoLISP evaluates all arguments before checking the argument types. In previous releases of AutoCAD, AutoLISP evaluated and checked the type of each argument sequentially. To see the difference, look at the following code examples:

```
(defun foo ()
  (print "Eval uati ng foo")
  '(1 2))

(defun bar ()
  (print "Eval uati ng bar")
  'b)

(defun baz ()
  (pri nt "Eval uati ng baz")
  'c)
```

Observe how an expression using the `inters` function is evaluated in AutoCAD 2000:

```
Command: (inters (foo) (bar) (baz))
"Evaluating foo"
"Evaluating bar"
"Evaluating baz"
; *** ERROR: too few arguments
```

Each argument was evaluated successfully before AutoLISP passed the results to `inters` and discovered that too few arguments were specified.

In AutoCAD R14, the same expression evaluated as follows:

```
Command: (inters (foo) (bar) (baz))
"Evaluating foo"
"Evaluating bar" error: bad argument type
```

AutoLISP evaluated `(foo)`, then passed the result to `inters`. Since the result was a valid 2D point list, AutoLISP proceeds to evaluate `(bar)`, where it determines that the evaluated result is a string, an invalid argument type for `inters`.

## Using the `*error*` Function

Proper use of the `*error*` function can ensure that AutoCAD returns to a particular state after an error occurs. Through this user-definable function you can assess the error condition and return an appropriate message to the user. If AutoCAD encounters an error during evaluation, it prints a message in the following form:

```
Error: text
```

In this message, *text* describes the error. However, if the `*error*` function is defined (that is, if it is not `nil`), AutoLISP executes `*error*` instead of printing the message. The `*error*` function receives *text* as its single argument.

If `*error*` is not defined or is `nil`, AutoLISP evaluation stops and displays a traceback of the calling function and its callers. It is beneficial to leave this error handler in effect while you debug your program.

A code for the last error is saved in the AutoCAD system variable `ERRNO`, where you can retrieve it by using the `getvar` function. See appendix C, “AutoLISP Error Codes,” for a list of error codes and their meaning.

Before defining your own `*error*` function, save the current contents of `*error*` so that the previous error handler can be restored upon exit. When an error condition exists, AutoCAD calls the currently defined `*error*` function and passes it one argument, which is a text string describing the nature of the error. Your `*error*` function should be designed to exit quietly after an `ESC` (cancel) or an `exit` function call. The standard way to accomplish this is to include the following statements in your error-handling routine.

```
(if  
  (or  
    (= msg "Function cancelled")  
    (= msg "quit / exit abort")  
  )  
  (princ)  
  (princ (strcat "\nError: " msg))  
)
```

This code examines the error message passed to it and ensures that the user is informed of the nature of the error. If the user cancels the routine while it is running, nothing is returned from this code. Likewise, if an error condition is programmed into your code and the `exit` function is called, nothing is returned. It is presumed you have already explained the nature of the error by using `print` statements. Remember to include a terminating call to `princ` if you don't want a return value printed at the end of an error routine.

The main caveat about error-handling routines is they are normal AutoLISP functions that can be canceled by the user. Keep them as short and as fast as possible. This will increase the likelihood that an entire routine will execute if called.

You can also warn the user about error conditions by displaying an alert box, which is a small dialog box containing a message supplied by your program. To display an alert box, call the `alert` function.

The following call to `alert` displays an alert box:

```
(alert "File not found")
```

## Catching Errors and Continuing Program Execution

Your program can intercept and attempt to process errors instead of allowing control to pass to `*error*`. The `vl-catch-all-apply` function is designed to invoke any function, return a value from the function, and trap any error that may occur. The function requires two arguments: a symbol identifying a function or `lambda` expression, and a list of arguments to be passed to the called function. The following example uses `vl-catch-all-apply` to divide two numbers:

```
_ $ (setq catchit (vl-catch-all-apply '/ '(50 5)))  
10
```

The result from this example is the same as if you had used `apply` to perform the division.

The value of `vl-catch-all-apply` is in catching errors and allowing your program to continue execution.

### To catch errors with `vl-catch-all-apply`

- 1 The following code defines a function named `catch-me-if-you-can`. Copy the code into a VLISP text editor window:

```
(defun catch-me-if-you-can (dividend divisor / errobj)  
  (setq errobj (vl-catch-all-apply '/ (list dividend divisor)))  
  (if (vl-catch-all-error-p errobj)  
      (progn  
        (print (strcat "An error occurred: "  
                      (vl-catch-all-error-message errobj)  
                      ))  
        )  
      (prompt "Do you want to continue? (Y/N) -> ")  
      (setq ans (getstring))  
      (if (equal (strcase ans) "Y")  
          (print "Okay, I'll keep going")  
          )  
      )  
  (print errobj)  
)  
(princ)
```

This function accepts two number arguments and uses `vl-catch-all-apply` to divide the first number by the second number. The `vl-catch-all-error-p` function determines whether the return value from `vl-catch-all-apply` is

an error object. If the return value is an error object, `catch-me-if-you-can` invokes `vl-catch-all-error-message` to obtain the message from the error object.

2 Load the function.

3 Invoke the function with the following command:

```
(catch-me-if-you-can 50 2)
```

The function should return 25.

4 Intentionally cause an error condition by invoking the function with the following command:

```
(catch-me-if-you-can 50 0)
```

The function should issue the following prompt in the AutoCAD Command window:

```
"An error occurred: divide by zero" Do you want to continue? (Y/N) ->
```

If you enter y, `catch-me-if-you-can` indicates that it will continue processing.

Try modifying this example by changing `vl-catch-all-apply` to `apply`. Load and run the example with a divide by zero again. When `apply` results in an error, execution immediately halts and `*error*` is called, resulting in an error message.

The `vl-catch-*` functions are especially important when you use ActiveX with AutoLISP. Many of the AutoCAD ActiveX automation methods are designed to be used in the “programming by exception” style. This means they either return useful values if they succeed, or raise an exception if they fail (instead of returning an error value). If your program uses ActiveX methods, you must prepare it to catch exceptions, otherwise the program halts, leaving the user at a Command prompt. See “Handling Errors Returned by ActiveX Methods” on page 191 for an example using `vl-catch-all-apply` with ActiveX.

# Using AutoLISP to Communicate with AutoCAD

AutoLISP provides various functions for examining the contents of the currently loaded drawing. This chapter introduces these functions and describes how to use them in conjunction with other functions.

## In This Chapter

# 9

- Query and Command Functions
- Display Control
- Getting User Input
- Geometric Utilities
- Conversions
- File Handling
- Device Access and Control

# Query and Command Functions

The query and command functions described in this section provide direct access to AutoCAD commands and drawing services. Their behavior depends on the current state of the AutoCAD system and environment variables, and on the drawing that is currently loaded. See “Query and Command Functions” in appendix A, “AutoLISP Function Synopsis,” for a complete list of query and command functions.

---

**NOTE** The AutoLISP examples in this chapter show code entered at the AutoCAD Command prompt, not the Visual LISP Console window.

---

## Command Submission

The `command` function sends an AutoCAD command directly to the AutoCAD Command prompt. The `command` function has a variable-length argument list. These arguments must correspond to the types and values expected by that command’s prompt sequence; these may be strings, real values, integers, points, entity names, or selection set names. Data such as angles, distances, and points can be passed either as strings or as the values themselves (as integer or real values, or as point lists). An empty string (“”) is equivalent to pressing the SPACEBAR or ENTER on the keyboard.

There are some restrictions on the commands that you can use with the `command` function. See the *AutoLISP Reference* definition of this function for information on these restrictions.

The following code fragment shows representative calls to `command`.

```
(command "circle" "0,0" "3,3")  
(command "thickness" 1)  
(setq p1 '(1.0 1.0 3.0))  
(setq rad 4.5)  
(command "circle" p1 rad)
```

If AutoCAD is at the Command prompt when these functions are called, AutoCAD performs the following actions:

- 1 The first call to `command` passes points to the CIRCLE command as strings (draws a circle centered at 0.0,0.0 and passes through 3.0,3.0).
- 2 The second call passes an integer to the THICKNESS command (changes the current thickness to 1.0).



- 3 The last call uses a 3D point and a real (floating-point) value, both of which are stored as variables and passed by reference to the CIRCLE command. This draws an extruded circle centered at (1.0,1.0,3.0) with a radius of 4.5.

## Foreign Language Support

If you develop AutoLISP programs that can be used with a foreign language version of AutoCAD, the standard AutoCAD commands and keywords are automatically translated if you precede each command or keyword with an underscore (\_).

```
(command "_line" pt1 pt2 pt3 "_c")
```

If you are using the dot prefix (to avoid using redefined commands), you can place the dot and underscore in either order. Both ".\_line" and "\_.line" are valid.

## Pausing for User Input

If an AutoCAD command is in progress and the predefined symbol PAUSE is encountered as an argument to `command`, the command is suspended to allow direct user input (usually point selection or dragging). This is similar to the backslash pause mechanism provided for menus.

The PAUSE symbol is defined as a string consisting of a single backslash.

When you use a backslash (\) in a string, you must precede it by another backslash (\\).

Menu input is not suspended by an AutoLISP pause. If a menu item is active when the `command` function pauses for input, that input request can be satisfied by the menu. If you want the menu item to be suspended as well, you must provide a backslash in the menu item. When valid input is found, both the `command` function and the menu item resume.

---

**NOTE** You can use a backslash instead of the PAUSE symbol. However, the pause mechanism might require a different trigger value in future versions of AutoLISP, so it is recommended that you always use the PAUSE symbol rather than an explicit backslash. Also, if the `command` function is invoked from a menu item, the backslash suspends the reading of the menu item, which results in partial evaluation of the AutoLISP expression.

---

If you issue a transparent command while a `command` function is suspended, the `command` function remains suspended. Therefore, users can 'ZOOM and 'PAN while at a `command` pause. The pause remains in effect until AutoCAD gets valid input, and no transparent command is in progress. For example, the following code begins the CIRCLE command, sets the center point at

(5,5), and then pauses to let the user drag the circle's radius. When the user specifies the desired point (or types in the desired radius), the function resumes, drawing a line from (5,5) to (7,5), as follows:

```
(command "circle" "5,5" pause "line" "5,5" "7,5" "")
```

If PAUSE is encountered when a command is expecting input of a text string or an attribute value, AutoCAD pauses for input only if the TEXTVAL system variable is nonzero. Otherwise, AutoCAD does not pause for user input but uses the value of the PAUSE symbol (a single backslash) text.

When the command function pauses for user input, the function is considered active, so the user cannot enter another AutoLISP expression to be evaluated.

The following is an example of using the PAUSE symbol (the layer NEW\_LAY and the block MY\_BLOCK must exist in the drawing prior to testing this code):

```
(setq blk "MY_BLOCK")
(setq old_lay (getvar "clayer"))
(command "layer" "set" "NEW_LAY" "")
(command "insert" blk pause "" "" pause)
(command "layer" "set" old_lay "")
```

The preceding code fragment sets the current layer to NEW\_LAY, pauses for user selection of an insertion point for the block MY\_BLOCK (which is inserted with *X* and *Y* scale factors of 1), and pauses again for user selection of a rotation angle. The current layer is then reset to the original layer.

If the command function specifies a PAUSE to the SELECT command and a PICKFIRST set is active, the SELECT command obtains the PICKFIRST set without pausing for the user.

---

**WARNING!** The Radius and Diameter subcommands of the Dim prompt issue additional prompts in some situations. This can cause a failure of AutoLISP programs written prior to Release 11 that use these commands.

---

## Passing Pick Points to AutoCAD Commands

Some AutoCAD commands (such as TRIM, EXTEND, and FILLET) require the user to specify a pick point as well as the object itself. To pass such pairs of object and point data by means of the command function without the use of a PAUSE, you must first store them as variables. Points can be passed as strings within the command function or can be defined outside the function and passed as variables, as shown in the following example. This code fragment shows one method of passing an entity name and a pick point to the command function.

<code>(command "circle" "5,5" "2")</code>	<i>Draws circle</i>
<code>(command "line" "3,5" "7,5" "")</code>	<i>Draws line</i>
<code>(setq el (entlast))</code>	<i>Gets last entity name</i>
<code>(setq pt '(5 7))</code>	<i>Sets point pt</i>
<code>(command "trim" el "" pt "")</code>	<i>Performs trim</i>

If AutoCAD is at the Command prompt when these functions are called, AutoCAD performs the following actions:

- 1 Draws a circle centered at (5,5) with a radius of 2.
- 2 Draws a line from (3,5) to (7,5).
- 3 Creates a variable `el` that is the name of the last object added to the database. (See “Using AutoLISP to Manipulate AutoCAD Objects” for more discussion of objects and object-handling functions.)
- 4 Creates a `pt` variable that is a point on the circle. (This point selects the portion of the circle to be trimmed.)
- 5 Performs the TRIM command by selecting the `el` object and by selecting the point specified by `pt`.

## Undoing Commands Issued with the command Function

An UNDO group is explicitly created around each command used with the `command` function. If a user enters U (or UNDO) after running an AutoLISP routine, only the last command will be undone. Additional entries of UNDO will step backward through the commands used in that routine. If you want a group of commands to be considered a group (or the entire routine), use the UNDO Begin and UNDO End options.

## System and Environment Variables

With the `getvar` and `setvar` functions, AutoLISP applications can inspect and change the value of AutoCAD system variables. These functions use a string to specify the variable name. The `setvar` function specifies a value of the type that the system variable expects. AutoCAD system variables come in various types: integers, real values, strings, 2D points, and 3D points. Values supplied as arguments to `setvar` must be of the expected type. If an invalid type is supplied, an AutoLISP error is generated.

The following code fragment ensures that subsequent FILLET commands use a radius of at least 1:

```
(if (< (getvar "filletrad") 1)
  (setvar "filletrad" 1)
)
```

See the *Command Reference* for a list and description of each AutoCAD system variable.

An additional function, `getenv`, provides AutoLISP routines with access to the currently defined operating system environment variables.

## Configuration Control

AutoCAD uses the *acadxx.cfg* file to store configuration information (the *xx* in the file name refers to the AutoCAD release number). The `AppData` section of this file is provided for users and developers to store configuration information pertaining to their applications. The `getcfg` and `setcfg` functions allow AutoLISP applications to inspect and change the value of parameters in the `AppData` section.

## Display Control

AutoLISP includes functions for controlling the AutoCAD display in both text and graphics windows. Some functions prompt for, or depend on, input from the AutoCAD user.

The `prompt`, `princ`, `prin1`, and `print` functions are the primary text output functions. These functions were described in the “AutoLISP Basics” chapter, under the heading, “Basic Output Functions.”

See “Display Control Functions” in appendix A, “AutoLISP Function Synopsis,” for a complete list of display control functions.

## Controlling Menus

The `menucmd` function controls the display of the graphics window menus. It displays, modifies, or queries one of the submenus of the current menu, and accepts a string argument that specifies the submenu and the action to perform on that submenu.

The `menucmd` function takes a string argument that consists of two fields, separated by an equal sign, in the following form:

```
"menu_area=action"
```

This syntax can load a submenu into a specified menu area, or perform an action on a menu item or a currently loaded menu area. The *menu\_area* field specifies which part of the menu is to receive the action. This field can specify a menu area, such as `PO` (for the shortcut menu) or `S` (for the screen menu), or a specific menu item. The *action* field specifies the action to perform on

the menu area or menu item, or a submenu to load into the menu area. The menu areas that can receive an action are the same as those used in menu file submenu references.

Every menu area has a currently loaded submenu. By default, the first submenu following a menu section label is loaded into that menu area.

If *menu\_area* specifies a pull-down menu or image tile menu, *action* can be an asterisk (\*). This causes the menu to display (pull-down menus and image tile menus are not automatically displayed when they are called). In Windows, only the P0 (cursor) menu and image tile menus are displayed with the asterisk.

---

**NOTE** Do not include the dollar sign that introduces the similar instructions in a menu file in the string argument. Also, do not include the asterisks that precede submenu labels in the menu file in the *action* field of the string argument.

---

The following `menucmd` function call causes the \*\*OSNAP screen submenu defined in the current menu file to be displayed (assuming the screen menu is currently enabled).

```
(menucmd "S=OSNAP")
```

In Windows, you can reference the menu group. This can be useful if there are multiple menus loaded that contain the same submenu name. The following code displays the \*\*OSNAP screen submenu in the ACAD menu group.

```
(menucmd "S=ACAD.OSNAP")
```

The `menucmd` function can load submenus into the BUTTONS and AUX menu areas. You might want your digitizer buttons to function differently depending on whether Tablet mode is on or off. You can have two submenus defined in the \*\*\*BUTTONS1 section, \*\*DIG-BUTTONS and \*\*TAB-BUTTONS, and switch between them with the following code.

```
(menucmd "B1=DIG-BUTTONS")  Enables the DIG-BUTTONS submenu  
(menucmd "B1=TAB-BUTTONS") Enables the TAB-BUTTONS submenu
```

The following code loads the \*\*\*POPO menu into the P0 (cursor) menu area and displays it.

```
(menucmd "PO=POPO")          Loads the ***POPO menu into the P0 menu area  
(menucmd "PO=*")             Displays it
```

If you are sure the correct menu is loaded into a particular menu area, you do not need to load it specifically each time you want to display it.

The following call displays the pull-down menu currently loaded in the P1 (first pull-down menu) location.

```
(menucmd "P1=*")
```

Using "P1=\*" without previously loading the menu can result in unexpected behavior. Although you can load virtually any menu at a pull-down or short-cut menu location, it is best to use only menus specifically designed for that menu area. For example, if you have a submenu called \*\*MORESTUFF, you can load it at the P1 location with the following code:

```
(menucmd "P1=MORESTUFF")    Loads the **MORESTUFF menu in the  
                             P1 menu location  
(menucmd "P1=*")           Displays it
```

This menu remains in this location until you replace it by loading another menu, as in the following:

```
(menucmd "P1=POP1")
```

If your menu uses the disabling (graying-out) and marking features, you can retrieve and change the state of a menu label with the `menucmd` function. The following call retrieves the current state of the fourth label in the pull-down menu P2.

```
(menucmd "P2. 4=#?" )      If disabled returns    "P2. 4=~"
```

These function calls enable and disable that same label:

```
(menucmd "P2. 4=")         Enables the label  
(menucmd "P2. 4=~")       Disables the label
```

You can also place and remove marks to the left of menu labels.

The previously described method of menu item handling works relatively well with a single static menu. However, it becomes unreliable when menu item locations change when you load multiple partial menu files. You can make use of the menu-group and name-tag features to keep track of menu items. Instead of specifying a menu item by its location in the menu file, you specify the menu group and name tag associated with the menu item.

When you use the menu group to enable, disable, and mark menu labels, you must precede the group name with a G, as shown in the following examples.

```
(menucmd "Gacad. ID_New=~") Disables the label  
(menucmd "Gacad. ID_New=")  Enables the label
```

Not only can an AutoLISP function enable and disable menu labels, it can also modify the text displayed in the label by placing a DIESEL string expression in the label. Because DIESEL accepts only strings as input, you can pass information to the DIESEL expression through a `USER$1-5` system variable that has been set to a value returned by your function.

You can also use the `menucmd` function to evaluate DIESEL string expressions within an AutoLISP function. The following routine returns the current time:

```
(defun C:CTIME ( / ctim)
  (setq ctim
    (menucmd "M=$(edtime,$(getvar,date),H:MMam/pm)"))
  (prin1 (strcat "\nThe current time is " ctim))
  (prin1))
)
```

For information on the use of DIESEL expressions with AutoLISP and a catalog of DIESEL functions, see the *Customization Guide*. Refer also to the *Customization Guide* for further information on menus.

## Control of Graphics and Text Windows

You can control the display of the graphics and text windows from an AutoLISP application. On single-screen AutoCAD installations, a call to `graphscr` displays the graphics window, and a call to `textscr` displays the text window. Using these functions is equivalent to toggling the Flip Screen function key. The function `textpage` is like `textscr`, but `textpage` clears the text window before displaying the next text window (as the AutoCAD STATUS and LIST commands do).

The `redraw` function is similar to the AutoCAD REDRAW command but provides more control over what is displayed. It not only redraws the entire graphics area but can also specify a single object to be redrawn or undrawn (that is, blanked out). If the object is a complex object such as an old-style polyline or a block, `redraw` can draw (or undraw) either the entire object or its header. The `redraw` function can also highlight and unhighlight specified objects.

## Control of Low-Level Graphics

AutoLISP provides functions that control the low-level graphics and allow direct access to the AutoCAD graphics screen and input devices.

The `grtext` function displays text directly in the status or menu areas, with or without highlighting. The `grdraw` function draws a vector in the current viewport with control over color and highlighting. The `grvecs` function draws multiple vectors.

**NOTE** Because these functions depend on code in AutoCAD, their operation can be expected to change from release to release. There is no guarantee that applications calling these functions will be upward compatible. Also, they depend on current hardware configurations. In particular, applications that call `grtext` are not likely to work the same on all configurations unless the developer is very careful to use them as described (see the *Customization Guide*) and to avoid hardware-specific features. Finally, because they are low-level functions, they do almost no error reporting and can alter the graphics screen display unexpectedly (see the following example for a way to fix this).

The following sequence restores the default graphics window display caused by incorrect calls to `grtext`, `grdraw`, or `grvecs`:

```
(grtext)           Restores standard text
(redraw)
```

# Getting User Input

Several functions enable an AutoLISP application to prompt the user for input of data. See “User Input Functions” in appendix A, “AutoLISP Function Synopsis,” for a complete list of user input functions.

## The getxxx Functions

Each user-input `getxxx` function pauses for data entry of the indicated type and returns the value entered. The application specifies an optional prompt to display before the function pauses. The following table lists the `getxxx` functions and the type of user input requested.

Allowable input to the getxxx user-input functions	
Function name	Type of user input
<code>getint</code>	An integer value on the Command line
<code>getreal</code>	A real or integer value on the Command line
<code>getstring</code>	A string on the Command line
<code>getpoint</code>	A point value on the Command line or selected from the screen
<code>getcorner</code>	A point value (the opposite corner of a box) on the Command line or selected from the screen



### Allowable input to the getxxx user-input functions (*continued*)

Function name	Type of user input
<code>getdist</code>	A real or integer value (of distance) on the Command line or determined by selecting points on the screen
<code>getangle</code>	An angle value (in the current angle format) on the Command line or based on selected points on the screen
<code>getorient</code>	An angle value (in the current angle format) on the Command line or based on selected points on the screen
<code>getkeyword</code>	A predefined keyword or its abbreviation on the Command line

**NOTE** Although the `getvar`, `getcfg`, and `getenv` functions begin with the letters *g*, *e*, and *t*, they are not user-input functions. They are discussed in “Query and Command Functions” on page 260.

The functions `getint`, `getreal`, and `getstring` pause for user input on the AutoCAD Command line. They return a value only of the same type as that requested.

The `getpoint`, `getcorner`, and `getdist` functions pause for user input on the Command line or from points selected on the graphics screen. The `getpoint` and `getcorner` functions return 3D point values, and `getdist` returns a real value.

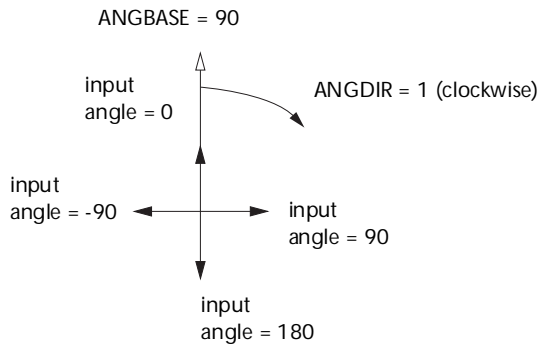
Both `getangle` and `getorient` pause for input of an angle value on the Command line or as defined by points selected on the graphics screen. For the `getorient` function, the 0 angle is always to the right: “East” or “3 o’clock.” For `getangle`, the 0 angle is the value of `ANGBASE`, which can be set to any angle. Both `getangle` and `getorient` return an angle value (a real) in radians measured counterclockwise from a base (0 angle), for `getangle` equal to `ANGBASE`, and for `getorient` to the right.

For example, `ANGBASE` is set to 90 degrees (north), and `ANGDIR` is set to 1 (clockwise direction for increasing angles). The following table shows what

`getangle` and `getorient` return (in radians) for representative input values (in degrees).

**Possible return values from `getangle` and `getorient`**

Input (degrees)	<code>getangle</code>	<code>getorient</code>
0	0.0	1.5708
-90	1.5708	3.14159
180	3.14159	4.71239
90	4.71239	0.0



The `getangle` function honors the settings of `ANGDIR` and `ANGBASE` when accepting input. You can use `getangle` to obtain a rotation amount for a block insertion, because input of 0 degrees always returns 0 radians. The `getorient` function honors only `ANGDIR`. You use `getorient` to obtain angles such as the baseline angle for a text object. For example, given the preceding settings of `ANGBASE` and `ANGDIR`, for a line of text created at an angle of 0, `getorient` returns an angle value of 90.

The user-input functions take advantage of the error-checking capability of AutoCAD. Trivial errors are trapped by AutoCAD and are not returned by the user-input function. A prior call to `initget` provides additional filtering capabilities, lessening the need for error-checking.

The `getkeyword` function pauses for the input of a keyword or its abbreviation. Keywords must be defined with the `initget` function before the call to `getkeyword`. All user-input functions (except `getstring`) can accept keyword

values in addition to the values they normally return, provided that `ini tget` has been called to define the keywords.

All user-input functions allow for an optional *prompt* argument. It is recommended you use this argument rather than a prior call to the `prompt` or `princ` functions. If a *prompt* argument is supplied with the call to the user-input function, that prompt is reissued in the case of invalid user input. If no *prompt* argument is supplied and the user enters incorrect information, the following message appears at the AutoCAD prompt line:

Try again:

This can be confusing, because the original prompt may have scrolled out of the Command prompt area.

The AutoCAD user cannot typically respond to a user-input function by entering an AutoLISP expression. If your AutoLISP routine makes use of the `ini tget` function, arbitrary keyboard input is permitted to certain functions that can allow an AutoLISP statement as response to a command implemented in AutoLISP. This is discussed in “Arbitrary Keyboard Input” on page 273.

## Control of User-Input Function Conditions

The `ini tget` function provides a level of control over the next user-input function call. The `ini tget` function establishes various options for use by the next `entsel`, `nentsel`, `nentsel p`, or `getxxx` function (except `getstring`, `getvar`, and `getenv`). This function accepts two arguments, *bits* and *string*, both of which are optional. The *bits* argument specifies one or more control bits that enable or disable certain input values to the following user-input function call. The *string* argument can specify keywords that the following user-input function call will recognize.

The control bits and keywords established by `ini tget` apply only to the next user-input function call. They are discarded after that call. The application doesn't have to call `ini tget` a second time to clear special conditions.

### Input Options for User-Input Functions

The value of the *bits* argument restricts the types of user input to the following user-input function call. This reduces error-checking. These are some of the available bit values: 1 disallows null input, 2 disallows input of 0 (zero), and 4 disallows negative input. If these values are used with a following call to the `getint` function, the user is forced to enter an integer value greater than 0.

To set more than one condition at a time, add the values together (in any combination) to create a *bits* value between 0 and 255. If *bits* is not included or is set to 0, none of the control conditions applies to the next user-input function call. (For a complete listing of *initget* bit settings, see “initget” in the *AutoLISP Reference*.)

```
(initget (+ 1 2 4))
(getint "\nHow old are you? ")
```

This sequence requests the user’s age. AutoCAD displays an error message and repeats the prompt if the user attempts to enter a negative or zero value, press ENTER only, or enter a string (the *getint* function rejects attempts to enter a value that is not an integer).

## Keyword Options

The optional *string* argument specifies a list of keywords recognized by the next user-input function call.

The *initget* function allows keyword abbreviations to be recognized in addition to the full keywords. The user-input function returns a predefined keyword if the input from the user matches the spelling of a keyword (not case sensitive), or if the user enters the abbreviation of a keyword. There are two methods for abbreviating keywords; both are discussed in the *initget* topic in the *AutoLISP Reference*.

The following user-defined function shows a call to *getreal*, preceded by a call to *initget*, that specifies two keywords. The application checks for these keywords and sets the input value accordingly.

```
(defun C:GETNUM (/ num)
  (initget 1 "Pi Two-pi")
  (setq num (getreal "Pi /Two-pi /<number>: "))
  (cond
    ((eq num "Pi ") pi)
    ((eq num "Two-pi ") (* 2.0 pi))
    (T num)
  )
)
```

This *initget* call inhibits null input (*bits* = 1) and establishes a list of two keywords, “Pi ” and “Two-pi ”. The *getreal* function is then used to obtain a real number, issuing the following prompt:

Pi/Two-pi/<number>:

The result is placed in local symbol *num*. If the user enters a number, that number is returned by *C:GETNUM*. However, if the user enters the keyword *Pi* (or simply *P*), *getreal* returns the keyword *Pi*. The *cond* function detects this

and returns the value of `p` in this case. The `Two-pi` keyword is handled similarly.

---

**NOTE** You can also use `initget` to enable `entsel`, `nentsel`, and `nentselp` to accept keyword input. For more information on these functions, see “Object Handling” on page 305 and the `entsel`, `nentsel`, and `nentselp` function definitions in the *AutoLISP Reference*.

---

## Arbitrary Keyboard Input

The `initget` function also allows arbitrary keyboard input to most `getxxx` functions. This input is passed back to the application as a string. An application using this facility can be written to permit the user to call an AutoLISP function at a `getxxx` function prompt.

These functions show a method for allowing AutoLISP response to a `getxxx` function call:

```
(defun C:ARBENTRY ( / pt1)
  (initget 128)                ; Sets arbitrary entry bit
  (setq pt1 (getpoint "\nPoint: ")) ; Gets value from user.
  (if (= 'STR (type pt1))      ; If it's a string, convert it
    (setq pt1 (eval (read pt1))) ; to a symbol, try evaluating
    ; it as a function; otherwise,
    ; just return the value.
    pt1
  )
)
(defun REF ( )
  (setvar "LASTPOINT" (getpoint "\nReference point: "))
  (getpoint "\nNext point: " (getvar "LASTPOINT"))
)
```

If both the `C:ARBENTRY` and `REF` functions are loaded into the drawing, the following command sequence is acceptable.

Command: **arbentry**

Point: **(ref)**

Reference point: *Select a point*

Next point: **@1,1,0**

## Input Validation

You should protect your code from unintentional user errors. The AutoLISP user input `getxxx` functions do much of this for you. However, it's dangerous to forget to check for adherence to other program requirements that the `getxxx` functions do not check for. If you neglect to check input validity, the program's integrity can be seriously affected.

# Geometric Utilities

A group of functions allows applications to obtain pure geometric information and geometric data from the drawing. See “Geometric Functions” in appendix A, “AutoLISP Function Synopsis,” for a complete list of geometric utility functions.

The `angle` function finds the angle in radians between a line and the *X* axis (of the current UCS), `distance` finds the distance between two points, and `polar` finds a point by means of polar coordinates (relative to an initial point). The `inters` function finds the intersection of two lines. The `osnap` and `textbox` functions are described separately.

The following code fragment shows calls to the geometric utility functions:

```
(setq pt1 '(3.0 6.0 0.0))
(setq pt2 '(5.0 2.0 0.0))
(setq base '(1.0 7.0 0.0))
(setq rads (angle pt1 pt2)) ; Angle in XY plane of current UCS
                           ; (value is returned in radians)
(setq len (distance pt1 pt2)) ; Distance in 3D space
(setq endpt (polar base rads len))
```

The call to `polar` sets `endpt` to a point that is the same distance from (1,7) as `pt1` is from `pt2`, and at the same angle from the *X* axis as the angle between `pt1` and `pt2`.

## Object Snap

The `osnap` function can find a point by using one of the AutoCAD Object Snap modes. The Snap modes are specified in a string argument.

The following call to `osnap` looks for the midpoint of an object near `pt1`:

```
(setq pt2 (osnap pt1 "mi dp"))
```

The following call looks for the midpoint, the endpoint, or the center of an object nearest `pt1`:

```
(setq pt2 (osnap pt1 "mi dp, endp, center"))
```

In both examples, `pt2` is set to the snap point if one is found that fulfills the `osnap` requirements. If more than one snap point fulfills the requirements, the point is selected based on the setting of the `SORTENTS` system variable. Otherwise, `pt2` is set to `nil`.

---

**NOTE** The APERTURE system variable determines the allowable proximity of a selected point to an object when you use Object Snap.

---

## Text Extents

The `textbox` function returns the diagonal coordinates of a box that encloses a text object. It takes an entity definition list of the type returned by `entget` (an association list of group codes and values) as its single argument. This list can contain a complete association list description of the text object or just a list describing the text string.

The points returned by `textbox` describe the bounding box (an imaginary box that encloses the text object) of the text object, as if its insertion point were located at (0,0,0) and its rotation angle were 0. The first list returned is the point (0.0 0.0 0.0), unless the text object is oblique or vertical or it contains letters with descenders (such as g and p). The value of the first point list specifies the offset distance from the text insertion point to the lower-left corner of the smallest rectangle enclosing the text. The second point list specifies the upper-right corner of that box. The returned point lists always describe the bottom-left and upper-right corners of this bounding box, regardless of the orientation of the text being measured.

The following example shows the minimum allowable entity definition list that `textbox` accepts. Because no additional information is provided, `textbox` uses the current defaults for text style and height.

Command: `(textbox '((1 . "Hello world"))  
((0.0 0.0 0.0) (2.80952 1.0 0.0))`

The actual values returned by `textbox` will vary depending on the current text style.

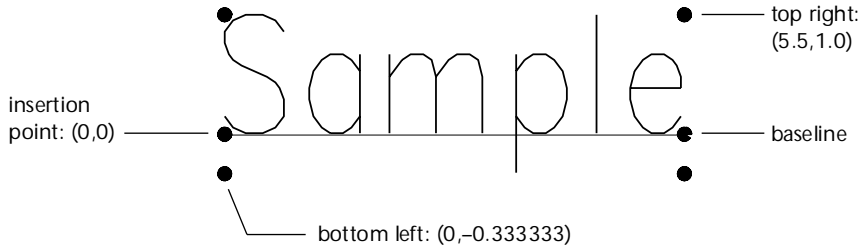
The following example demonstrates one method of providing the `textbox` function with an entity definition list.

```
Command: dtext
Justify/Style/<Start point>: 1,1
Height <1.0000>: ENTER
Rotation angle <0>: ENTER
Text: test
Text: ENTER
```

```
Command: (setq e (entget (entlast)))
((-1 . <Entity name: 1ba3568>) (0 . "TEXT") (330 . <Entity name: 1ba34f8>) (5 .
"2D") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 .
"AcDbText") (10 1.0 1.0 0.0) (40 . 1.0) (1 . "test") (50 . 0.0) (41 . 1.0) (51
. 0.0) (7 . "Standard") (71 . 0) (72 . 0) (11 0.0 0.0 0.0) (210 0.0 0.0 1.0)
(100 . "AcDbText") (73 . 0))
```

```
Command: (textbox e)
((0.0 0.0 0.0) (0.8 0.2 0.0))
```

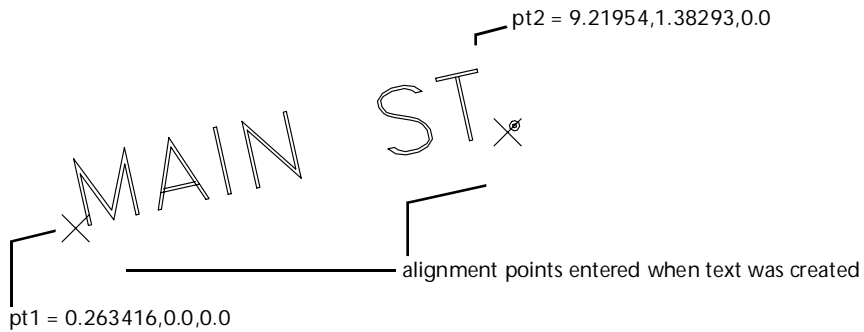
The following figure shows the results of applying `textbox` to a text object with a height of 1.0. The figure also shows the baseline and insertion point of the text.



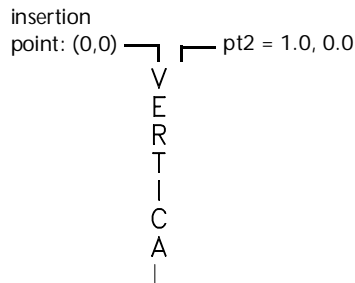
If the text is vertical or rotated, `pt1` is still the bottom-left corner and `pt2` is the upper-right corner; the bottom-left point may have negative offsets if necessary.

The following figure shows the point values (`pt1` and `pt2`) that `textbox` returns for samples of vertical and aligned text. In both samples, the height of the letters is 1.0. (For the aligned text, the height is adjusted to fit the alignment points.)





When using vertical text styles, the points are still returned in left-to-right, bottom-to-top order as they are for horizontal styles, so that the first point list will contain negative offsets from the text insertion point.



$pt1 = -0.5, -20.0$

Regardless of the text orientation or style, the points returned by `textbox` are such that the text insertion point (group code 10) directly translates to the origin point of the object coordinate system (OCS) for the associated text object. This point can be referenced when translating the coordinates returned from `textbox` into points that define the actual extent of the text. The two sample routines that follow use `textbox` to place a box around selected text regardless of its orientation.

The first routine uses the `textbox` function to draw a box around a selected text object:

```
(defun C:TBOX ( / textent tb ll ur ul lr)
  (setq textent (car (entsel "\nSelect text: ")))
  (command "ucs" "Object" textent)
  (setq tb (textbox (list (cons -1 textent)))
        ll (car tb)
        ur (cadr tb)
        ul (list (car ll) (cadr ur))
        lr (list (car ur) (cadr ll))
  )
  (command "pline" ll lr ur ul "Close")
  (command "ucs" "p")
  (princ)
)
```

The second routine, which follows, accomplishes the same task as the first routine by performing the geometric calculations with the `sin` and `cos` AutoLISP functions. The result is correct only if the current UCS is parallel to the plane of the text object.

```
(defun C:TBOX2 ( / textent ang sinrot cosrot
                  t1 t2 p0 p1 p2 p3 p4)
  (setq textent (entget (car (entsel "\nSelect text: "))))
  (setq p0 (cdr (assoc 10 textent))
        ang (cdr (assoc 50 textent))
        sinrot (sin ang)
        cosrot (cos ang)
        t1 (car (textbox textent))
        t2 (cadr (textbox textent))
        p1 (list
            (+ (car p0)
              (- (* (car t1) cosrot) (* (cadr t1) sinrot))
            )
            (+ (cadr p0)
              (+ (* (car t1) sinrot) (* (cadr t1) cosrot))
            )
          )
        p2 (list
            (+ (car p0)
              (- (* (car t2) cosrot) (* (cadr t1) sinrot))
            )
            (+ (cadr p0)
              (+ (* (car t2) sinrot) (* (cadr t1) cosrot))
            )
          )
        p3 (list
            (+ (car p0)
              (- (* (car t2) cosrot) (* (cadr t2) sinrot))
            )
            (+ (cadr p0)
              (+ (* (car t2) sinrot) (* (cadr t2) cosrot))
            )
          )
  )
  (command "pline" p1 p2 p3 p4 "Close")
  (command "ucs" "p")
  (princ)
)
```

```

p4 (list
  (+ (car p0)
    (- (* (car t1) cosrot)(* (cadr t2) sinrot))
  )
  (+ (cadr p0)
    (+ (* (car t1) sinrot)(* (cadr t2) cosrot))
  )
)
)
(command "pline" p1 p2 p3 p4 "c")
(princ)
)

```

## Conversions

The functions described in this section are utilities for converting data types and units. See “Conversion Functions” in appendix A, “AutoLISP Function Synopsis,” for a complete list of conversion functions.

### String Conversions

The functions `rtos` (real to string) and `angtos` (angle to string) convert numeric values used in AutoCAD to string values that can be used in output or as textual data. The `rtos` function converts a real value, and `angtos` converts an angle. The format of the result string is controlled by the value of AutoCAD system variables: the units and precision are specified by `LUNITS` and `LUPREC` for real (linear) values and by `AUNITS` and `AUPREC` for angular values. For both functions, the dimensioning variable `DIMZIN` controls how leading and trailing zeros are written to the result string.

The following code fragments show calls to `rtos` and the values returned (assuming the `DIMZIN` system variable equals 0). Precision (the third argument to `rtos`) is set to 4 places in the first call and 2 places in the others.

```

(setq x 17.5)
(setq str "\nValue formatted as ")

(setq fmtval (rtos x 1 4)) ; Mode 1 = scientific
(princ (strcat str fmtval)) ; displays Value formatted as 1.7500E+01

(setq fmtval (rtos x 2 2)) ; Mode 2 = decimal
(princ (strcat str fmtval)) ; displays Value formatted as 17.50

(setq fmtval (rtos x 3 2)) ; Mode 3 = engineering
(princ (strcat str fmtval)) ; displays Value formatted as 1'-5.50"

(setq fmtval (rtos x 4 2)) ; Mode 4 = architectural
(princ (strcat str fmtval)) ; displays Value formatted as 1'-5 1/2"

(setq fmtval (rtos x 5 2)) ; Mode 5 = fractional
(princ (strcat str fmtval)) ; displays Value formatted as 17 1/2

```

When the UNITMODE system variable is set to 1, specifying that units are displayed as entered, the string returned by `rtos` differs for engineering (mode equals 3), architectural (mode equals 4), and fractional (mode equals 5) units. For example, the first two lines of the preceding sample output would be the same, but the last three lines would appear as follows:

Value formatted as 1'5.50"  
 Value formatted as 1'5-1/2"  
 Value formatted as 17-1/2"

Because the `angtos` function takes the ANGBASE system variable into account, the following code always returns "0":

```
(angtos (getvar "angbase"))
```

There is no AutoLISP function that returns a string version (in the current mode/precision) of either the amount of rotation of ANGBASE from true zero (East) or an arbitrary angle in radians.

To find the amount of rotation of ANGBASE from AutoCAD zero (East) or the size of an arbitrary angle, you can do one of the following:

- Add the desired angle to the current ANGBASE, and then check to see if the absolute value of the result is greater than  $2\pi$  ( $2 * \pi$ ). If so, subtract  $2\pi$ ; if the result is negative, add  $2\pi$ , then use the `angtos` function on the result.
- Store the value of ANGBASE in a temporary variable, set ANGBASE to 0, evaluate the `angtos` function, then set ANGBASE to its original value.

Subtracting the result of `(atof (angtos 0))` from 360 degrees ( $2\pi$  radians or 400 grads) also yields the rotation of ANGBASE from 0.

The `distof` (distance to floating point) function is the complement of `rtos`. Therefore, the following calls, which use the strings generated in the previous examples, all return the same value: 17.5. (Note the use of the backslash (\) with modes 3 and 4.)

```
(distof "1.7500E+01" 1) ; Mode 1 = scientific
(distof "17.50" 2) ; Mode 2 = decimal
(distof "1'-5.50\" 3) ; Mode 3 = engineering
(distof "1'-5 1/2\" 4) ; Mode 4 = architectural
(distof "17 1/2" 5) ; Mode 5 = fractional
```

The following code fragments show similar calls to `angtos` and the values returned (still assuming that DIMZIN equals 0). Precision (the third argument to `angtos`) is set to 0 places in the first call, 4 places in the next three calls, and 2 places in the last.

```

(setq ang 3.14159 str2 "\nAngle formatted as ")

(setq fmtval (angtos ang 0 0)) ; Mode 0 = degrees
(princ (strcat str2 fmtval)) ; displays Angle formatted as 180

(setq fmtval (angtos ang 1 4)) ; Mode 1 = deg/mi n/sec
(princ (strcat str2 fmtval)) ; displays Angle formatted as 180d0'0"

(setq fmtval (angtos ang 2 4)) ; Mode 2 = grads
(princ (strcat str2 fmtval)) ; displays Angle formatted as 200.0000g

(setq fmtval (angtos ang 3 4)) ; Mode 3 = radians
(princ (strcat str2 fmtval)) ; displays Angle formatted as 3.1416r

(setq fmtval (angtos ang 4 2)) ; Mode 4 = surveyor's
(princ (strcat str2 fmtval)) ; displays Angle formatted as W

```

The UNITMODE system variable also affects strings returned by `angtos` when it returns a string in surveyor's units (mode equals 4). If UNITMODE equals 0, the string returned can include spaces (for example, "N 45d E"); if UNITMODE equals 1, the string contains no spaces (for example, "N45dE").

The `angtof` function complements `angtos`, so all of the following calls return the same value: 3.14159.

```

(angtof "180" 0) ; Mode 0 = degrees
(angtof "180d0'0\" 1) ; Mode 1 = deg/mi n/sec
(angtof "200.0000g" 2) ; Mode 2 = grads
(angtof "3.14159r" 3) ; Mode 3 = radians
(angtof "W" 4) ; Mode 4 = surveyor's

```

When you have a string specifying a distance in feet and inches, or an angle in degrees, minutes, and seconds, you must precede the quotation mark with a backslash (\) so it doesn't look like the end of the string. The preceding examples of `angtof` and `distof` demonstrate this action.

## Angular Conversion

If your application needs to convert angular values from radians to degrees, you can use the `angtos` function, which returns a string, and then convert that string into a floating point value with `atof`.

```

(setq pt1 '(1 1) pt2 '(1 2))
(setq rad (angle pt1 pt2))
(setq deg (atof (angtos rad 0 2))) returns 90.0

```

However, a more efficient method might be to include a `Radian->Degrees` function in your application. The following code shows this:

```

; Convert value in radians to degrees
(defun Radian->Degrees (nbrOfRadians)
  (* 180.0 (/ nbrOfRadians pi)))
)

```

After this function is defined, you can use the Radi an->Degrees function throughout your application, as in

```
(setq degrees (Radi an->Degrees rad)) returns 90.0
```

You may also need to convert from degrees to radians. The following code shows this:

```
; Convert value in degrees to radians
(defun Degrees->Radians (numberOfDegrees)
  (* pi (/ numberOfDegrees 180.0))
) ;_ end of defun
```

## ASCII Code Conversion

AutoLISP provides the `ascii` and `chr` functions that handle decimal ASCII codes. The `ascii` function returns the ASCII decimal value associated with a string, and `chr` returns the character associated with an ASCII decimal value.

To see your system's characters with their codes in decimal, octal, and hexadecimal form, save the following AutoLISP code to a file named *ascii.lsp*. Then load the file and enter the new ASCII command at the AutoCAD Command prompt. This command prints the ASCII codes to the screen and to a file called *ascii.txt*. The C: ASCII function makes use of the BASE function. You may find this conversion utility useful in other applications.

```
; BASE converts from a decimal integer to a string in another base.
(defun BASE ( bas int / ret yyy zot )
  (defun zot ( i1 i2 / xxx )
    (if (> (setq xxx (rem i2 i1)) 9)
      (chr (+ 55 xxx))
      (itoa xxx))
  )
  (setq ret (zot bas int) yyy (/ int bas))
  (while (>= yyy bas)
    (setq ret (strcat (zot bas yyy) ret))
    (setq yyy (/ yyy bas))
  )
  (strcat (zot bas yyy) ret)
)

(defun C:ASCII ( / chk out ct code dec oct hex )
  (initget "Yes")
  (setq chk (getkeyword "\nWriting to ASCII.TXT, continue? <Y>: "))
  (if (or (= chk "Yes") (= chk nil)) (progn
    (setq out (open "ascii.txt" "w") chk 1 code 0 ct 0)
    (princ "\n\nCHAR DEC OCT HEX\n")
    (princ "\n\nCHAR DEC OCT HEX\n" out)
    (while chk
      (setq dec (strcat " " (itoa code))
        oct (base 8 code) hex (base 16 code))
      (setq dec (substr dec (- (strlen dec) 2) 3))
```

)  
(pri nc)  
)

# Unit Conversion

The *acad.unt* file defines various conversions between real-world units such as miles to kilometers, Fahrenheit to Celsius, and so on. The function `cvunit` takes a value expressed in one system of units and returns the equivalent value in another system. The two systems of units are specified by strings containing expressions of units defined in *acad.unt*.

The `convert` function does not convert incompatible dimensions. For example, it does not convert inches into grams.

The first time `cvunit` converts to or from a unit during a drawing editor session, it must look up the string that specifies the unit in *acad.unit*. If your application has many values to convert from one system of units to another, it is more efficient to convert the value 1.0 by a single call to `cvunit` and then use the returned value as a scale factor in subsequent conversions. This works for all units defined in *acad.unit*, except temperature scales, which involve an offset as well as a scale factor.

## Converting from Inches to Meters

If the current drawing units are engineering or architectural (feet and inches), the following routine converts a user-specified distance of inches into meters:

```
(defun C:IM ( / eng_len metric_len eng metric)
  (princ "\nConverting inches to meters. ")
  (setq eng_len
    (getdist "\nEnter a distance in inches: "))
  (setq metric_len (cvunit eng_len "inches" "meters"))
  (setq eng (rtos eng_len 2 4)
    metric (rtos metric_len 2 4))
  (princ
    (strcat "\n\t" eng " inches = " metric " meters. "))
  (princ)
)
```

## The Unit Definition File

With the AutoCAD unit definition file *acad.unit*, you can define factors to convert data in one set of units to another set of units. The definitions in *acad.unit* are in ASCII format and are used by the unit-conversion function *cvunit*.

You can make new units available by using a text editor to add their definitions to *acad.unit*. A definition consists of two lines in the file—the unit name and the unit definition. The first line must have an asterisk (\*) in the first column, followed by the name of the unit. A unit name can have several abbreviations or alternate spellings, separated by commas. If a unit name has singular and plural forms, you can specify these using the following format:

```
*[ [common] [ ( [singular.] plural ) ] ]...
```

You can specify multiple expressions (singular and plural). They don't have to be located at the end of the word, and a plural form isn't required. The following are examples of valid unit name definitions:

```
*inch(es)
*milleni (um. a)
*foot. eet) or (foot. feet)
```

The line following the *\*unit name* line defines the unit as either fundamental or derived.

### Fundamental Units

A fundamental unit is an expression in constants. If the line following the *\*unit name* line begins with something other than an equal sign (=), it defines fundamental units. Fundamental units consist of five integers and two real numbers in the following form:

```
c, e, h, k, m, r1, r2
```



The five integers correspond to the exponents of these five constants:

c	Velocity of light in a vacuum
e	Electron charge
h	Planck's constant
k	Boltzman's constant
m	Electron rest mass

As a group, these exponents define the dimensionality of the unit: length, mass, time, volume, and so on.

The first real number (r1) is a multiplier, and the second (r2) is an additive offset (used only for temperature conversions). The fundamental unit definition allows for different spellings of the unit (for example, *meter* and *metre*); the case of the unit is ignored. An example of a fundamental unit definition is as follows:

```
*meter(s), metre(s), m
-1, 0, 1, 0, -1, 4. 1214856408e11, 0
```

In this example, the constants that make one meter are as follows:

$$\left( \frac{1}{c} \times h \times \frac{1}{m} \right) \times (4.1214856 \times 10^{11})$$

### Derived Units

A derived unit is defined in terms of other units. If the line following the *\*unit name* line begins with an equal sign (=), it defines derived units. Valid operators in these definitions are \* (multiplication), / (division), + (addition), - (subtraction), and ^ (exponentiation). You can specify a predefined unit by naming it, and you can use abbreviations (if provided). The items in a formula are multiplied together unless some other arithmetic operator is specified. For example, the units database defines the dimensionless multiple and submultiple names, so you can specify a unit such as micro-inches by entering `mi cro i nch`. The following are examples of derived unit definitions.

```
; Units of area
*township(s)
=93239571. 456 meter^2
```

The definition of a township is given as 93,239,571.456 square meters.

```
; Electromagnetic units
```

```
*volt(s), v  
=watt/ampere
```

In this example, a volt is defined as a watt divided by an ampere. In the *acad.unt*, both watts and amperes are defined in terms of fundamental units.

### User Comments

To include comments, begin the line with a semicolon. The comment continues to the end of the line.

```
; This entire line is a comment.
```

List the *acad.unt* file itself for more information and examples.

## Coordinate System Transformations

The *trans* function translates a point or a displacement from one coordinate system into another. It takes a point argument, *pt*, that can be interpreted as either a 3D point or a 3D displacement vector, distinguished by a displacement argument called *disp*. The *disp* argument must be nonzero if *pt* is to be treated as a displacement vector; otherwise, *pt* is treated as a point. A *from* argument specifies the coordinate system in which *pt* is expressed, and a *to* argument specifies the desired coordinate system. The following is the syntax for the *trans* function:

```
(trans pt from to [disp])
```

The following AutoCAD coordinate systems can be specified by the *from* and *to* arguments:

WCS	World coordinate system—the reference coordinate system. All other coordinate systems are defined relative to the WCS, which never changes. Values measured relative to the WCS are stable across changes to other coordinate systems.
UCS	User coordinate system—the working coordinate system. The user specifies a UCS to make drawing tasks easier. All points passed to AutoCAD commands, including those returned from AutoLISP routines and external functions, are points in the current UCS (unless the user precedes them with a * at the Command prompt). If you want your application to send coordinates in the WCS, OCS, or DCS to AutoCAD commands, you must first convert them to the UCS by calling the <i>trans</i> function.

OCS	Object coordinate system—point values returned by <code>entget</code> are expressed in this coordinate system, relative to the object itself. These points are usually converted into the WCS, current UCS, or current DCS, according to the intended use of the object. Conversely, points must be translated into an OCS before they are written to the database by means of the <code>entmod</code> or <code>entmake</code> functions. This is also known as the entity coordinate system.
DCS	<p>Display coordinate system—the coordinate system into which objects are transformed before they are displayed. The origin of the DCS is the point stored in the AutoCAD system variable <code>TARGET</code>, and its <i>Z</i> axis is the viewing direction. In other words, a viewport is always a plan view of its DCS. These coordinates can be used to determine where something will be displayed to the AutoCAD user.</p> <p>When the <i>from</i> and <i>to</i> integer codes are 2 and 3, in either order, 2 indicates the DCS for the current model space viewport and 3 indicates the DCS for paper space (PSDCS). When the 2 code is used with an integer code other than 3 (or another means of specifying the coordinate system), it is assumed to indicate the DCS of the current space, whether paper space or model space. The other argument is also assumed to indicate a coordinate system in the current space.</p>
PSDCS	Paper space DCS—this coordinate system can be transformed <i>only</i> to or from the DCS of the currently active model space viewport. This is essentially a 2D transformation, where the <i>X</i> and <i>Y</i> coordinates are always scaled and are offset if the <i>disp</i> argument is 0. The <i>Z</i> coordinate is scaled but is never translated. Therefore, it can be used to find the scale factor between the two coordinate systems. The PSDCS (integer code 2) can be transformed only into the current model space viewport. If the <i>from</i> argument equals 3, the <i>to</i> argument must equal 2, and vice versa.

Both the *from* and *to* arguments can specify a coordinate system in any of the following ways:

- As an integer code that specifies the WCS, current UCS, or current DCS (of either the current viewport or paper space).
- As an entity name returned by one of the entity name or selection set functions described in “Using AutoLISP to Manipulate AutoCAD Objects.” This specifies the OCS of the named object. For planar objects, the OCS can differ from the WCS, as described in the AutoCAD *User’s Guide*. If the OCS does not differ, conversion between OCS and WCS is an identity operation.
- As a 3D extrusion vector. Extrusion vectors are always represented in World coordinates; an extrusion vector of (0,0,1) specifies the WCS itself.

The following table lists the valid integer codes that can be used as the *to* and *from* arguments:

Coordinate system codes	
Code	Coordinate system
0	World (WCS)
1	User (current UCS)
2	Display; DCS of current viewport when used with code 0 or 1, DCS of current model space viewport when used with code 3
3	Paper space DCS, PSDCS (used only with code 2)

The following example translates a point from the WCS into the current UCS.

```
(setq pt '(1.0 2.0 3.0))  
(setq cs_from 0) ; WCS  
(setq cs_to 1) ; UCS  
(trans pt cs_from cs_to 0) ; disp = 0 indicates that pt is a point
```

If the current UCS is rotated 90 degrees counterclockwise around the World Z axis, the call to *trans* returns a point (2.0,-1.0,3.0). However, if you swap the *to* and *from* values, the result differs as shown in the following code:

```
(trans pt cs_to cs_from 0) ; the result is (-2.0 1.0 3.0)
```

## Point Transformations

If you are doing point transformations with the `trans` function and you need to make that part of a program run faster, you can construct your own transformation matrix on the AutoLISP side by using `trans` once to transform each of the basis vectors (0 0 0), (1 0 0), (0 1 0), and (0 0 1). Writing matrix multiplication functions in AutoLISP can be difficult, so it may not be worthwhile unless your program is doing a lot of transformations.

# File Handling

AutoLISP provides functions for handling files and data I/O. See “File-Handling Functions” in appendix A, “AutoLISP Function Synopsis,” for a complete list of file-handling functions.

## File Search

An application can use the `findfile` function to search for a particular file name. The application can specify the directory to search, or it can use the current AutoCAD library path.

In the following code fragment, `findfile` searches for the requested file name according to the AutoCAD library path:

```
(setq refname "refc.dwg")
(setq fil (findfile refname))
(if fil
  (setq refname fil)
  (princ (strcat "\nCoul d not find file " refname ". " )))
)
```

If the call to `findfile` is successful, the variable `refname` is set to a fully qualified path name string, as follows:

```
"/home/work/ref/refc.dwg"
```

When specifying a path name, you must precede the backslash (\) with another backslash so the path name will be recognized by AutoLISP. Alternatively, you can use the slash character (/) as a directory separator.

The `getfiled` function displays a dialog box containing a list of available files of a specified extension type in the specified directory. This gives AutoLISP routines access to the AutoCAD Get File dialog box.

A call to `getfiled` takes four arguments that determine the appearance and functionality of the dialog box. The application must specify the following string values, each of which can be nil: a title, placed at the top of the dialog

box; a default file name, displayed in the edit box at the bottom of the dialog box; and an extension type, which determines the initial files provided for selection in the list box. The final argument is an integer value that specifies how the dialog box interacts with selected files.

This simple routine uses `getfiled` to let you view your directory structure and select a file:

```
(defun C:DDIR ( )
  (setq dfile (getfiled "Directory Listing" "" "" 2))
  (princ (strcat "\nVariable 'dfile' set to selected file " dfile ))
  (princ)
)
```

This is a useful utility command. The `dfile` variable is set to the file you select, which can then be used by other AutoLISP functions or as a response to a Command line prompt for a file name. To use this variable in response to a Command line prompt, enter **!dfile**.

---

**NOTE** You cannot use **!dfile** in a dialog box. It is valid only at the command line.

---

For more information, see “getfiled” in the *AutoLISP Reference*.

## Accessing Help Files

The `help` function provides access to both AutoCAD Help files (*.ahp*) and Windows Help files (*.hlp*). Depending on the Help file’s extension, the `help` function calls the AutoCAD or the Windows Help viewer with the specified file. You can use this function to add a Help facility to your applications. The following code fragment calls the default AutoCAD Help file and provides information about the `LINE` command.

```
(help "" "line")
```

You can create a Help file that provides information about your applications or about procedures you use in your business. The following user-defined command displays the *morehelp.hlp* Help file as follows:

```
(defun C:MYHELP ( )
  (help "morehelp.hlp")
  (princ)
)
```

See the *Customization Guide* for information on creating and modifying help files.

The `setfunhelp` function provides help for user-defined commands. After the definition of your new command, adding a call to `setfunhelp` associates

a specific help file and topic with that command. The following example assigns the help topic “Mycmd” in the file *morehelp.hlp* to the user-defined MYCMD command:

```
(defun C:MYCMD ( )  
  . Command definition  
  .  
)  
(setfunhelp c:mycmd "morehelp.hlp" "mycmd")
```

## Device Access and Control

AutoLISP provides the `grread` and `tablet` functions for accessing data from the various input devices.

Note that the `read-char` and `read-line` file-handling functions can also read input from the keyboard input buffer. See the *AutoLISP Reference* for more information on these functions.

### Accessing User Input

The `grread` function returns raw user input, whether from the keyboard or from the pointing device (mouse or digitizer). If the call to `grread` enables tracking, the function returns a digitized coordinate that can be used for things such as dragging.

---

**NOTE** There is no guarantee that applications calling `grread` will be upward compatible. Because it depends on the current hardware configuration, applications that call `grread` are not likely to work in the same way on all configurations.

---

### Calibrating Tablets

AutoCAD users can calibrate a digitizing tablet by using the `TABLET` command (see the *Command Reference* for a description of this command). The `tablet` function enables applications to manage calibration by setting the calibrations directly and by saving those settings for future use.

The first argument to the `tablet` function is an integer *code*. If *code* is equal to 0, the function returns the current calibration. If *code* is equal to 1, the calibration is set according to the remaining arguments. Calibrations are expressed as four 3D points (in addition to the *code*). The first three points—*row1*, *row2*, and *row3*—are the three rows of the tablet’s transformation

matrix. The fourth point, *direction*, is a vector that is normal to the plane in which the tablet's surface is assumed to lie (expressed in WCS, the World Coordinate System). When the calibration is set with the **TABLET** command, the tablet's surface is assumed to lie in the *XY* plane of the current UCS.

---

**NOTE** The **TABMODE** system variable controls whether Tablet mode is turned on (1) or off (0). You can control it by using the **setvar** function.

---

The following sample routine retrieves the current tablet calibration and stores it in the variable **tcal** :

```
(defun C:TABGET ( )
  (setq tcal (tablet 0))
  (if tcal
    (princ
      (strcat "\nConfiguration saved, "
        "use TABSET to retrieve calibration.")
    )
    (princ "\nCalibration not obtainable ")
  )
  (princ)
)
```

If the preceding routine was successful, the symbol **tcal** now contains the list returned by the **tablet** function. This list might appear as follows:

```
(1 (0.00561717 -0.000978942 -7.5171)
  (0.000978942 0.00561717 -9.17308)
  (0.0 0.0 1.0)
  (0.0 0.0 1.0)
)
```

To reset the calibration to the values retrieved by the preceding routine, you can use the **C: TABSET** routine, as follows:

```
(defun C:TABSET ( )
  (if (not (apply 'tablet tcal))
    (princ "\nUnable to reset calibration. ")
    (progn
      (princ "\nTablet calibration reset. ")
      (setvar "tabmode" 1)
      (if (= (getvar "tabmode") 0)
        (princ "\nUnable to turn on tablet mode ")
      )
    )
  )
  (princ)
)
```

The transformation matrix passed as *row1*, *row2*, and *row3* is a 3×3 transformation matrix that is meant to transform a 2D point. The 2D point is



expressed as a column vector in homogeneous coordinates (by appending 1.0 as the third element), so the transformation looks like this:

$$\begin{bmatrix} X' \\ Y' \\ D' \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & 1.0 \end{bmatrix} \bullet \begin{bmatrix} X \\ Y \\ 1.0 \end{bmatrix}$$

The calculation of a point is similar to the 3D case. AutoCAD transforms the point by using the following formulas:

$$X' = M_{00}X + M_{01}Y + M_{02}$$

$$Y' = M_{10}X + M_{11}Y + M_{12}$$

$$D' = M_{20}X + M_{21}Y + 1.0$$

To turn the resulting vector back into a 2D point, the first two components are divided by the third component (the scale factor  $D'$ ) yielding the point  $(X'/D', Y'/D')$ .

For projective transformations, the most general case, `tablet` does the full calculation. But for affine and orthogonal transformations,  $M_{20}$  and  $M_{21}$  are both 0, so  $D'$  would be 1.0. The calculation of  $D'$  and the division are omitted; the resulting 2D point is simply  $(X', Y')$ .

As the previous paragraph implies, an affine transformation is a special, uniform case of a projective transformation. An orthogonal transformation is a special case of an affine transformation: not only are  $M_{20}$  and  $M_{21}$  zero, but  $M_{00} = M_{11}$  and  $M_{10} = -M_{01}$ .

---

**NOTE** When you set a calibration, the list returned does not equal the list provided if the *direction* isn't normalized. AutoCAD normalizes the direction vector before it returns it. Also, it ensures the third element in the third column (`row3[Z]`) is equal to 1. This situation should not arise if you set the calibration by using values retrieved from AutoCAD by means of `tablet`. However, it can happen if your program calculates the transformation itself.

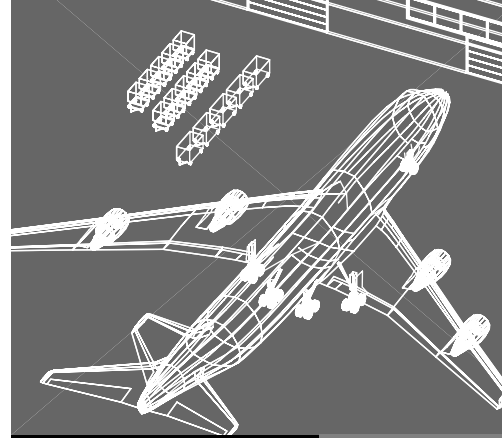
---



# Using AutoLISP to Manipulate AutoCAD Objects

Most AutoLISP functions that handle selection sets and objects identify a set or an object by the entity name. For selection sets, which are valid only in the current session, the volatility of names poses no problem, but it does for objects because they are saved in the drawing database. An application that must refer to the same objects in the same drawing (or drawings) at different times can use the objects' handle.

AutoLISP uses symbol tables to maintain lists of graphic and non-graphic data related to a drawing, such as the layers, linetypes, and block definitions. Each symbol table entry has a related entity name and handle and can be manipulated in a manner similar to the way other AutoCAD entities are manipulated.



## In This Chapter

# 10

- Selection Set Handling
- Object Handling
- Extended Data—xdata
- Xrecord Objects
- Symbol Table and Dictionary Access

# Selection Set Handling

AutoLISP provides a number of functions for handling selection sets. For a complete list of selection set functions, see “Selection Set Manipulation Functions” in appendix A, “AutoLISP Function Synopsis.”

The `ssget` function provides the most general means of creating a selection set. It can create a selection set in one of the following ways:

- Explicitly specifying the objects to select, using the Last, Previous, Window, Implied, WPolygon, Crossing, CPolygon, or Fence options
- Specifying a single point
- Selecting the entire database
- Prompting the user to select objects

With any option, you can use filtering to specify a list of attributes and conditions that the selected objects must match.

---

**NOTE** Selection set and entity names are volatile. That is, they apply only to the current drawing session.

---

The first argument to `ssget` is a string that describes which selection option to use. The next two arguments, *pt1* and *pt2*, specify point values for the relevant options (they should be left out if they don't apply). A point list, *pt-list*, must be provided as an argument to the selection methods that allow selection by polygons (that is, Fence, Crossing Polygon, and Window Polygon). The last argument, *filter-list*, is optional. If *filter-list* is supplied, it specifies the list of entity field values used in filtering. For example, you can obtain a selection set that includes all objects of a given type, on a given layer, or of a given color. Selection filters are described in more detail in “Selection Set Filter Lists” on page 298.

See the `ssget` entry in the *AutoLISP Reference* for a list of the available selection methods and the arguments used with each.

The following table shows examples of calls to `ssget`:

SSGET Examples	
Function call	Effect
<code>(setq pt1 '(0.0 0.0 0.0) pt2 '(5.0 5.0 0.0) pt3 '(4.0 1.0 0.0) pt4 '(2.0 6.0 0.0))</code>	Sets <code>pt1</code> , <code>pt2</code> , <code>pt3</code> , and <code>pt4</code> to point values
<code>(setq ss1 (ssget))</code>	Asks the user for a general object selection and places those items in a selection set
<code>(setq ss1 (ssget "P"))</code>	Creates a selection set from the most recently created selection set
<code>(setq ss1 (ssget "L"))</code>	Creates a selection set of the last object added to the database that is visible on the screen
<code>(setq ss1 (ssget pt2))</code>	Creates a selection set of an object passing through point (5,5)
<code>(setq ss1 (ssget "W" pt1 pt2))</code>	Creates a selection set of the objects inside the window from (0,0) to (5,5)
<code>(setq ss1 (ssget "F" (list pt2 pt3 pt4)))</code>	Creates a selection set of the objects crossing the fence and defined by the points (5,5), (4,1), and (2,6)
<code>(setq ss1 (ssget "WP" (list pt1 pt2 pt3)))</code>	Creates a selection set of the objects inside the polygon defined by the points (0,0), (5,5), and (4,1)
<code>(setq ss1 (ssget "X"))</code>	Creates a selection set of all objects in the database

When an application has finished using a selection set, it is important to release it from memory. You can do this by setting it to `nil`:

```
(setq ss1 nil)
```

Attempting to manage a large number of selection sets simultaneously is not recommended. An AutoLISP application cannot have more than 128 selection sets open at once. (The limit may be lower on your system.) When the limit is reached, AutoCAD refuses to create more selection sets. Keep a minimum number of sets open at a time, and set unneeded selection sets to `nil`.

as soon as possible. If the maximum number of selection sets is reached, you must call the `gc` function to free unused memory before another `ssget` will work.

## Selection Set Filter Lists

An entity filter list is an association list that uses DXF group codes in the same format as a list returned by `entget`. (See the *DXF Reference* for a list of group codes.) The `ssget` function recognizes all group codes except entity names (group -1), handles (group 5), and xdata codes (groups greater than 1000). If an invalid group code is used in a *filter-list*, it is ignored by `ssget`. To search for objects with xdata, use the -3 code as described in “Filtering for Extended Data” on page 300.

When a *filter-list* is provided as the last argument to `ssget`, the function scans the selected objects and creates a selection set containing the names of all main entities matching the specified criteria. For example, you can obtain a selection set that includes all objects of a given type, on a given layer, or of a given color.

The *filter-list* specifies which property (or properties) of the entities are to be checked and which values constitute a match.

The following examples demonstrate methods of using a *filter-list* with various object selection options.

SSGET examples using filter lists	
Function call	Effect
(setq ss1 (ssget ' ((0 . "TEXT")))) )	Prompts for general object selection but adds only text objects to the selection set.
(setq ss1 (ssget "P" ' ((0 . "LINE"))) )	Creates a selection set containing all line objects from the last selection set created.
(setq ss1 (ssget "W" pt1 pt2 ' ((8 . "FLOOR9"))) )	Creates a selection set of all objects inside the window that are also on layer FLOOR9.
(setq ss1 (ssget "X" ' ((0 . "CIRCLE"))) )	Creates a selection set of all objects in the database that are Circle objects.

## SSGET examples using filter lists (continued)

Function call	Effect
<pre>(ssget "l" ' ((0 . "LINE") (62 . 5)))</pre>	Creates a selection set of all blue Line objects that are part of the Implied selection set (those objects selected while PICKFIRST is in effect). Note that this filter picks up lines that have been assigned color 5 (blue), but not blue lines that have had their color applied by the ByLayer or ByBlock properties.

If both the code and the desired value are known, the list may be quoted as shown previously. If either is specified by a variable, the list must be constructed using the `list` and `cons` function. For example, the following code creates a selection set of all objects in the database that are on layer FLOOR3:

```
(setq lay_name "FLOOR3")
(setq ss1
  (ssget "X"
    (list (cons 8 lay_name))
  )
)
```

If the *filter-list* specifies more than one property, an entity is included in the selection set only if it matches all specified conditions, as in the following example:

```
(ssget "X" (list (cons 0 "CIRCLE")(cons 8 lay_name)(cons 62 1)))
```

This code selects only Circle objects on layer FLOOR3 that are colored red. This type of test performs a Boolean “AND” operation. Additional tests for object properties are described in “Logical Grouping of Filter Tests” on page 302.

The `ssget` function filters a drawing by scanning the selected entities and comparing the fields of each main entity against the specified filtering list. If an entity’s properties match all specified fields in the filtering list, it is included in the returned selection set. Otherwise, the entity is not included in the selection set. The `ssget` function returns `nil` if no entities from those selected match the specified filtering criteria.

**NOTE** The meaning of certain group codes can differ from entity to entity, and not all group codes are present in all entities. If a particular group code is specified in a filter, entities not containing that group code are excluded from the selection set that `ssget` returns.

When `ssget` filters a drawing, the selection set it retrieves might include entities from both paper space and model space. However, when the selection set is passed to an AutoCAD command, only entities from the space that is currently in effect are used. (The space to which an entity belongs is specified by the value of its 67 group. Refer to the *Customization Guide* for further information.)

## Wild-Card Patterns in Filter Lists

Symbol names specified in filtering lists can include wild-card patterns. The wild-card patterns recognized by `ssget` are the same as those recognized by the `wcmatch` function, and are described in “Wild-Card Matching” on page 238, and under “`wcmatch`” in the *AutoLISP Reference*.

When filtering for anonymous blocks, you must precede the `*` character with a reverse single quotation mark (```), also known as an escape character, because the `*` is read by `ssget` as a wild-card character. For example, you can retrieve an anonymous block named `*U2` with the following:

```
(ssget "X" ' ((2 . "`*U2")))
```

## Filtering for Extended Data

Using the `ssget filter-list`, you can select all entities containing extended data for a particular application. (See “Extended Data—`xdata`” on page 323.) To do this, use the `-3` group code, as shown in the following example:

```
(ssget "X" ' ((0 . "CIRCLE") (-3 ("APPNAME"))))
```

This code will select all circles that include extended data for the “APPNAME” application. If more than one application name is included in the `-3` group’s list, an AND operation is implied and the entity must contain extended data for all of the specified applications. So, the following statement would select all circles with extended data for both the “APP1” and “APP2” applications:

```
(ssget "X" ' ((0 . "CIRCLE") (-3 ("APP1") ("APP2"))))
```

Wild-card matching is permitted, so either of the following statements will select all circles with extended data for either or both of these applications.

```
(ssget "X" ' ((0 . "CIRCLE") (-3 ("APP[12]"))))  
(ssget "X" ' ((0 . "CIRCLE") (-3 ("APP1, APP2"))))
```

## Relational Tests

Unless otherwise specified, an equivalency is implied for each item in the `filter-list`. For numeric groups (integers, reals, points, and vectors), you can specify other relations by including a special `-4` group code that specifies a relational operator. The value of a `-4` group is a string indicating the test operator to be applied to the next group in the `filter-list`.



The following selects all circles with a radius (group code 40) greater than or equal to 2.0:

```
(ssget "X" ' ((0 . "CIRCLE") (-4 . ">=") (40 . 2.0)))
```

The possible relational operators are shown in the following table:

Relational operators for selection set filter lists	
Operator	Description
"*"	Anything goes (always true)
"="	Equals
"!="	Not equal to
" / ="	Not equal to
"<>"	Not equal to
"<"	Less than
"<="	Less than or equal to
">"	Greater than
">="	Greater than or equal to
"&"	Bitwise AND (integer groups only)
"&="	Bitwise masked equals (integer groups only)

The use of relational operators depends on the kind of group you are testing:

- All relational operators except for the bitwise operators ("&" and "&=") are valid for both real- and integer-valued groups.
- The bitwise operators "&" and "&=" are valid only for integer-valued groups. The bitwise AND, "&", is true if  $((integer\_group \& filter) \neq 0)$ —that is, if any of the bits set in the mask are also set in the integer group. The bitwise masked equals, "&=", is true if  $((integer\_group \& filter) = filter)$ —that is, if all bits set in the mask are also set in the *integer\_group* (other bits might be set in the *integer\_group* but are not checked).
- For point groups, the *X*, *Y*, and *Z* tests can be combined into a single string, with each operator separated by commas (for example, ">, >, \*"). If

- an operator is omitted from the string (for example, "=", "<>" leaves out the Z test), then the "anything goes" operator, "\*", is assumed.
- Direction vectors (group type 210) can be compared only with the operators "\*", "=", and "!=" (or one of the equivalent "not equal" strings).
  - You cannot use the relational operators with string groups; use wild-card tests instead.

## Logical Grouping of Filter Tests

You can also test groups by creating nested Boolean expressions that use the logical grouping operators shown in the following table:

Grouping operators for selection set filter lists		
Starting operator	Encloses	Ending operator
"<AND"	One or more operands	"AND>"
"<OR"	One or more operands	"OR>"
"<XOR"	Two operands	"XOR>"
"<NOT"	One operand	"NOT>"

The grouping operators are specified by -4 groups, like the relational operators. They are paired and must be balanced correctly in the filter list or the ssget call will fail. An example of grouping operators in a filter list follows:

```
(ssget "X"
' (
  (-4 . "<OR")
    (-4 . "<AND")
      (0 . "CIRCLE")
      (40 . 1.0)
    (-4 . "<AND>")
      (-4 . "<AND")
        (0 . "LINE")
        (8 . "ABC")
      (-4 . "<AND>")
    (-4 . "OR>")
  )
)
```

This code selects all circles with a radius of 1.0 plus all lines on layer "ABC". The grouping operators are not case-sensitive; for example, you can specify "and", "<or", instead of "AND>", "<OR".

Grouping operators are not allowed within the `-3` group. Multiple application names specified in a `-3` group use an implied AND operator. If you want to test for extended data using other grouping operators, specify separate `-3` groups and group them as desired. To select all circles having extended data for either application "APP1" or "APP2" but not both, enter the following:

```
(ssget "X"
' (0 . "CIRCLE")
  (-4 . "<XOR")
    (-3 ("APP1"))
    (-3 ("APP2"))
  (-4 . "XOR>")
)
```

You can simplify the coding of frequently used grouping operators by setting them equal to a symbol. The previous example could be rewritten as follows (notice that in this example you must explicitly quote each list):

```
(setq <xor '(-4 . "<XOR")
      xor> '(-4 . "XOR>") )
(ssget "X"
  (list
    ' (0 . "CIRCLE")
    <xor
      ' (-3 ("APP1"))
      ' (-3 ("APP2"))
    xor>
  )
)
```

As you can see, this method may not be sensible for short pieces of code but can be beneficial in larger applications.

## Selection Set Manipulation

Once a selection set has been created, you can add entities to it or remove entities from it with the functions `ssadd` and `ssdel`. You can use the `ssadd` function to create a new selection set, as shown in the following example. The following code fragment creates a selection set that includes the first and last entities in the current drawing (`entnext` and `entlast` are described later in this chapter).

```
(setq fname (entnext))           ; Gets first entity in the
                                ; drawing.
(setq lname (entlast))           ; Gets last entity in the
                                ; drawing.
```

```

(if (not fname)
  (princ "\nNo entities in drawing. ")
  (progn
    (setq ourset (ssadd fname)) ; Creates a selection set of the
                                ; first entity.
    (ssadd lname ourset)       ; Adds the last entity to the
                                ; selection set.
  )
)

```

The example runs correctly even if only one entity is in the database (in which case both `entnext` and `entlast` set their arguments to the same entity name). If `ssadd` is passed the name of an entity already in the selection set, it ignores the request and does not report an error. The following function removes the first entity from the selection set created in the previous example:

```
(ssdel fname ourset)
```

If there is more than one entity in the drawing (that is, if `fname` and `lname` are not equal), then the selection set `ourset` contains only `lname`, the last entity in the drawing.

The function `ssl length` returns the number of entities in a selection set, and `ssmemb` tests whether a particular entity is a member of a selection set. Finally, the function `ssname` returns the name of a particular entity in a selection set, using an index to the set (entities in a selection set are numbered from 0).

The following code shows calls to `ssname`:

```

(setq sset (ssget)) ; Prompts the user to create a
                    ; selection set.
(setq ent1 (ssname sset 0)) ; Gets the name of the first
                             ; entity in sset.
(setq ent4 (ssname sset 3)) ; Gets the name of the fourth
                             ; entity in sset.
(if (not ent4)
  (princ "\nNeed to select at least four entities. ")
)
(setq ilast (ssl length sset)) ; Finds index of the last entity
                              ; in sset.
                              ; Gets the name of the
                              ; last entity in sset.
(setq lastent (ssname sset (1- ilast)))

```

Regardless of how entities are added to a selection set, the set never contains duplicate entities. If the same entity is added more than once, the later additions are ignored. Therefore, `ssl length` accurately returns the number of distinct entities in the specified selection set.

## Passing Selection Sets between AutoLISP and ObjectARX Applications

When passing selection sets between AutoLISP and ObjectARX applications, the following should be observed:

If a selection set is created in AutoLISP and stored in an AutoLISP variable, then overwritten by a value returned from an ObjectARX application, the original selection set is eligible for garbage collection (it is freed at the next automatic or explicit garbage collection).

This is true even if the value returned from the ObjectARX application was the original selection set. In the following example, if the `adsfunc` ObjectARX function returns the same selection set it was fed as an argument, then this selection set will be eligible for garbage collection even though it is still assigned to the same variable.

```
(setq var1 (ssget))  
(setq var1 (adsfunc var1))
```

If you want the original selection set to be protected from garbage collection, then you must not assign the return value of the ObjectARX application to the AutoLISP variable that already references the selection set. Changing the previous example prevents the selection set referenced by `var1` from being eligible for garbage collection.

```
(setq var1 (ssget))  
(setq var2 (adsfunc var1))
```

## Object Handling

AutoLISP provides functions for handling objects. The object-handling functions are organized into two categories: functions that retrieve the entity name of a particular object, and functions that retrieve or modify entity data. See “Object-Handling Functions” in appendix A, “AutoLISP Function Synopsis,” for a complete list of the object-handling functions.

### Entity Name Functions

To operate on an object, an AutoLISP application must obtain its entity name for use in subsequent calls to the entity data or selection set functions. Two functions described in this section, `entsel` and `nentsel`, return not only the entity’s name but additional information for the application’s use.

Both functions require the AutoCAD user to select an object interactively by picking a point on the graphics screen. All the other entity name functions can retrieve an entity even if it is not visible on the screen or if it is on a frozen layer. The `entsel` function prompts the user to select an object by picking a point on the graphics screen, and `entsel` returns both the entity name and the value of the point selected. Some entity operations require knowledge of the point by which the object was selected. Examples from the set of existing AutoCAD commands include: `BREAK`, `TRIM`, and `EXTEND`. The `nentsel` function is described in detail in “Entity Context and Coordinate Transform Data” on page 307. These functions accept keywords if they are preceded by a call to `initget`.

The `entnext` function retrieves entity names sequentially. If `entnext` is called with no arguments, it returns the name of the first entity in the drawing database. If its argument is the name of an entity in the current drawing, `entnext` returns the name of the succeeding entity.

The following code fragment illustrates how `ssadd` can be used in conjunction with `entnext` to create selection sets and add members to an existing set.

```
(setq e1 (entnext))
(if (not e1)                                ; Sets e1 to name of first entity.
    (princ "\nNo entities in drawing. ")
    (progn
      (setq ss (ssadd))                    ; Sets ss to a null selection set.
      (ssadd e1 ss)                        ; Returns selection set ss with
                                          ; e1 added.
      (setq e2 (entnext e1))              ; Gets entity following e1.
      (ssadd e2 ss)                        ; Adds e2 to selection set ss.
    )
)
```

The `entlast` function retrieves the name of the last entity in the database. The last entity is the most recently created main entity, so `entlast` can be called to obtain the name of an entity that has just been created with a call to `command`.

You can set the entity name returned by `entnext` to the same variable name passed to this function. This “walks” a single entity name variable through the database, as shown in the following example:

```
(setq one_ent (entnext))                    ; Gets name of first entity.

(while one_ent
  .
  .                                         ; Processes new entity.
  .
  (setq one_ent (entnext one_ent))
)                                           ; Value of one_ent is now nil.
```

## Entity Handles and Their Uses

The `handent` function retrieves the name of an entity with a specific handle. As with entity names, handles are unique within a drawing. However, an entity's handle is constant throughout its life. AutoLISP applications that manipulate a specific database can use `handent` to obtain the current name of an entity they must use. You can use the `DDMODIFY` command to get the handle of a selected object.

The following code fragment uses `handent` to obtain and display an entity name.

```
(if (not (setq e1 (handent "5a2")))
  (princ "\nNo entity with that handle exists. ")
  (princ e1)
)
```

In one particular editing session, this code fragment might display the following:

<Entity name: 60004722>

In another editing session with the same drawing, the fragment might display an entirely different number. But in both cases the code would be accessing the same entity.

The `handent` function has an additional use. Entities deleted from the database (with `entdel`, described in the following section) are not purged until the current drawing ends. This means that `handent` can recover the names of deleted entities, which can then be restored to the drawing by a second call to `entdel`.

---

**NOTE** Handles are provided for block definitions, including subentities.

---

Entities in drawings that are cross-referenced by way of XREF Attach are not actually part of the current drawing; their handles are unchanged but cannot be accessed by `handent`. However, when drawings are combined by means of `INSERT`, `INSERT *`, `XREF Bind (XBIND)`, or `partial DXFIN`, the handles of entities in the incoming drawing are lost, and incoming entities are assigned new handle values to ensure each handle in the current drawing remains unique.

## Entity Context and Coordinate Transform Data

The `nentsel` and `nentselp` functions are similar to `entsel`, except they return two additional values to handle entities nested within block references.

Another difference between these functions is that when the user responds to a `nentsel` call by selecting a complex entity or a complex entity is selected by `nentsel p`, these functions return the entity name of the selected subentity and not the complex entity's header, as `entsel` does.

For example, when the user selects a 3D polyline, `nentsel` returns a vertex subentity instead of the polyline header. To retrieve the polyline header, the application must use `entnext` to step forward to the `seend` subentity, and then obtain the name of the header from the `seend` subentity's `-2` group. The same applies when the user selects attributes in a nested block reference.

Selecting an attribute within a block reference returns the name of the attribute and the pick point. When the selected object is a component of a block reference other than an attribute, `nentsel` returns a list containing the following elements:

- The selected entity's name.
- A list containing the coordinates of the point used to pick the object.
- The Model to World Transformation Matrix. This is a list consisting of four sublists, each of which contains a set of coordinates. This matrix can be used to transform the entity definition data points from an internal coordinate system called the model coordinate system (MCS), to the World Coordinate System (WCS). The insertion point of the block that contains the selected entity defines the origin of the MCS. The orientation of the UCS when the block is created determines the direction of the MCS axes.
- A list containing the entity name of the block that contains the selected object. If the selected object is in a nested block (a block within a block), the list also contains the entity names of all blocks in which the selected object is nested, starting with the innermost block and continuing outward until the name of the block that was inserted in the drawing is reported.

The list returned from selecting a block with `nentsel` is summarized as follows:

```
((<Entity Name: ename1>      ; Name of entity.
  (Px Py Pz)                ; Pick point.
  ( (X0 Y0 Z0)               ; Model to World Transformation Matrix.
    (X1 Y1 Z1)
    (X2 Y2 Z2)
    (X3 Y3 Z3)
  )
  (<Entity name: ename2>    ; Name of most deeply nested block
    .                      ; containing selected object.
    .
    .
  (<Entity name: enamen>)  ; Name of outermost block
  )                      ; containing selected object.
```



In the following example, create a block to use with the nentsel function.

```
Command: line
Specify first point: 1,1
Specify next point or [Undo]: 3,1
Specify next point or [Undo]: 3,3
Specify next point or [Close/Undo]: 1,3
Specify next point or [Close/Undo]: c
Command: -block
Enter block name or [?]: square
Specify insertion base point: 2,2
Select objects: Select the four lines you just drew
Select objects: ENTER
```

Then, insert the block in a UCS rotated 45 degrees about the Z axis:

```
Command: ucs
Current ucs name: *WORLD*
Enter option[New/Move/orthoGraphic/Prev/Restore/Save/Del/Apply/?/World]
<World>: z
Specify rotation angle about Z axis <0>: 45
Command: -insert
Enter block name or [?]: square
Specify insertion point or [Scale/X/Y/Z/Rotate/PScale/PX/PY/PZ/PRotate]: 7,0
Enter X scale factor, specify opposite corner, or [Corner/XYZ] <1>: ENTER
Enter Y scale factor <use X scale factor>: ENTER
Specify rotation angle <0>: ENTER
```

Use nentsel to select the lower-left side of the square.

```
(setq ndata (nentsel))
```

This code sets ndata equal to a list similar to the following:

```
(<Entity Name: 400000a0> ; Entity name.
(6.46616 -1.0606 0.0) ; Pick point.
((0.707107 0.707107 0.0) ; Model to World
(-0.707107 0.707107 0.0) ; Transformation Matrix.
(0.0 -0.0 1.0)
(4.94975 4.94975 0.0)
)
(<Entity name: 6000001c>) ; Name of block containing
; selected object.
)
```

Once you obtain the entity name and the Model to World Transformation Matrix, you can transform the entity definition data points from the MCS to the WCS. Use entget and assoc on the entity name to obtain the definition points expressed in MCS coordinates. The Model to World Transformation

Matrix returned by `nentsel` is a  $4 \times 3$  matrix—passed as an array of four points—that uses the convention that a point is a row rather than a column. The transformation is described by the following matrix multiplication:

$$\begin{bmatrix} X' & Y' & Z' & 1.0 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1.0 \end{bmatrix} \bullet \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \\ M_{30} & M_{31} & M_{32} \end{bmatrix}$$

So the equations for deriving the new coordinates are as follows:

$$X' = XM_{00} + YM_{10} + ZM_{20} + M_{30}$$

$$Y' = XM_{01} + YM_{11} + ZM_{21} + M_{31}$$

$$Z' = XM_{02} + YM_{12} + ZM_{22} + M_{32}$$

The  $M_{ij}$ , where  $0 \leq i, j \leq 2$ , are the Model to World Transformation Matrix coordinates;  $X, Y, Z$  is the entity definition data point expressed in MCS coordinates, and  $X', Y', Z'$  is the resulting entity definition data point expressed in WCS coordinates.

To transform a vector rather than a point, do not add the translation vector ( $M_{30} M_{31} M_{32}$  from the fourth column of the transformation matrix).

---

**NOTE** This is the only AutoLISP function that uses a matrix of this type. The `nentselp` function is preferred to `nentsel` because it returns a matrix similar to those used by other AutoLISP and ObjectARX functions.

---

Using the entity name previously obtained with `nentsel`, the following example illustrates how to obtain the MCS start point of a line (group code 10) contained in a block definition:

Command: `(setq edata (assoc 10 (entget (car ndata))))`  
`(10 -1.0 1.0 0.0)`

The following statement stores the Model to World Transformation Matrix sublist in the symbol `matrix`.

Command: `(setq matrix (caddr ndata))`  
`((0.707107 0.707107 0.0) ; X transformation`  
`(-0.707107 0.707107 0.0) ; Y transformation`  
`(0.0 -0.0 1.0) ; Z transformation`  
`(4.94975 4.94975 0.0) ; Displacement from WCS origin`  
`)`

The following command applies the transformation formula for  $X'$  to change the  $X$  coordinate of the start point of the line from an MCS coordinate to a WCS coordinate:

```
(setq answer
  (+
    (* (car (nth 0 matrix)) (cadr edata)) ; add:
    (* (car (nth 1 matrix)) (caddr edata)) ; M00 * X
    (* (car (nth 2 matrix)) (caddr edata)) ; M10 * Y
    (car (nth 3 matrix)) ; M20 * Z
  )
)
```

This statement returns 3.53553, the WCS  $X$  coordinate of the start point of the selected line.

## Entity Access Functions

The entity access functions are relatively slow. It is best to get the contents of a particular entity (or symbol table entry) once and keep that information stored in memory, rather than repeatedly ask AutoCAD for the same data. Be sure the data remains valid. If the user has an opportunity to alter the entity or symbol table entry, you should reissue the entity access function to ensure the validity of the data.

## Entity Data Functions

The functions described in this section operate on entity data and can be used to modify the current drawing database.

### Deleting an Entity

The `entdel` function deletes a specified entity. The entity is not purged from the database until the end of the current drawing session, so if the application calls `entdel` a second time during that session and specifies the same entity, the entity is undeleted.

Attributes and old-style polyline vertices cannot be deleted independently of their parent entities. The `entdel` function operates only on main entities. If you need to delete an attribute or vertex, you can use `command` to invoke the AutoCAD `ATTEDIT` or `PEDIT` commands.

### Obtaining Entity Information

The `entget` function returns the definition data of a specified entity. The data is returned as a list. Each item in the list is specified by a DXF group code. The first item in the list contains the entity's current name.

In this example, the following (default) conditions apply to the current drawing:

- Layer is 0
- Linetype is CONTINUOUS
- Elevation is 0

The user has drawn a line with the following sequence of commands:

Command: **line**  
From point: **1,2**  
To point: **6,6**  
To point: **ENTER**

An AutoLISP application can retrieve and print the definition data for the line by using the following AutoLISP function:

```
(defun C:PRINTDXF ( )  
  (setq ent (entlast)) ; Set ent to last entity.  
  (setq entl (entget ent)) ; Set entl to association list of  
                           ; last entity.  
  (setq ct 0) ; Set ct (a counter) to 0.  
  (textpage) ; Switch to the text screen.  
  (princ "\nentget of last entity:")  
  (repeat (length entl) ; Repeat for number of members in list:  
    (print (nth ct entl)) ; Print a newline, then each list  
                          ; member.  
    (setq ct (1+ ct)) ; Increments the counter by one.  
  )  
  (princ) ; Exit quietly.  
)
```

This would print the following:

```
entget of last entity:  
(-1 . <Entity name: 1bbd1c8>)  
(0 . "LINE")  
(330 . <Entity name: 1bbd0c8>)  
(5 . "69")  
(100 . "AcDbEntity")  
(67 . 0)  
(410 . "Model")  
(8 . "0")  
(100 . "AcDbLine")  
(10 1.0 2.0 0.0)  
(11 6.0 6.0 0.0)  
(210 0.0 0.0 1.0)
```

The -1 item at the start of the list contains the name of the entity. The `entmod` function, which is described in this section, uses the name to identify the entity to be modified. The individual dotted pairs that represent the values can be extracted by using `assoc` with the `cdr` function.

Sublists for points are not represented as dotted pairs like the rest of the values returned. The convention is that the `cdr` of the sublist is the group's value. Because a point is a list of two or three reals, the entire group is a three- (or four-) element list. The `cdr` of the group is the list representing the point, so the convention that `cdr` always returns the value is preserved.

The codes for the components of the entity are those used by DXF. As with DXF, the entity header items (color, linetype, thickness, the attributes-follow flag, and the entity handle) are returned only if they have values other than the default. Unlike DXF, optional entity definition fields are returned whether or not they equal their defaults and whether or not associated X, Y, and Z coordinates are returned as a single point variable, rather than as separate X (10), Y (20), and Z (30) groups.

All points associated with an object are expressed in terms of that object's object coordinate system (OCS). For point, line, 3D line, 3D face, 3D polyline, 3D mesh, and dimension objects, the OCS is equivalent to the WCS (the object points are World points). For all other objects, the OCS can be derived from the WCS and the object's extrusion direction (its 210 group). When working with objects that are drawn using coordinate systems other than the WCS, you may need to convert the points to the WCS or to the current UCS by using the `trans` function.

When writing functions to process entity lists, make sure the function logic is independent of the order of the sublists; use `assoc` to guarantee this. The `assoc` function searches a list for a group of a specified type. The following code returns the object type "LINE" (0) from the list `entl`.

```
(cdr (assoc 0 entl))
```

If the DXF group code specified is not present in the list (or if it is not a valid DXF group), `assoc` returns `nil`.

---

**WARNING!** Before performing an `entget` on vertex entities, you should read or write the polyline entity's header. If the most recently processed polyline entity is different from the one to which the vertex belongs, width information (the 40 and 41 groups) can be lost.

---

## Modifying an Entity

The `entmod` function modifies an entity. It passes a list that has the same format as a list returned by `entget` but with some of the entity group values (presumably) modified by the application. This function complements `entget`. The primary mechanism by which an AutoLISP application updates the database is by retrieving an entity with `entget`, modifying its entity list, and then passing the list back to the database with `entmod`.

The following code fragment retrieves the definition data of the first entity in the drawing and changes its layer property to `MYLAYER`.

```
(setq en (entnext))      ; Sets en to first entity name
                          ; in the drawing.
(setq ed (entget en))    ; Sets ed to the entity data
                          ; for entity name en.
(setq ed
  (subst (cons 8 "MYLAYER")
    (assoc 8 ed)          ; Changes the layer group in ed.
    ed)                  ; to layer MYLAYER.
  )
)
(entmod ed)              ; Modifies entity en's layer in
                          ; the drawing.
```

There are restrictions on the changes to the database that `entmod` can make; `entmod` cannot change the following:

- The entity's type or handle.
- Internal fields. (Internal fields are the values that AutoCAD assigns to certain group codes: -2, entity name reference; -1, entity name; 5, entity handle.) Any attempt to change an internal field—for example, the main entity name in a second subentity (group -2)—is ignored.
- Viewport entities. An attempt to change a viewport entity causes an error.

Other restrictions apply when modifying dimensions and hatch patterns.

AutoCAD must recognize all objects (except layers) that the entity list refers to. The name of any text style, linetype, shape, or block that appears in an entity list must be defined in the current drawing before the entity list is passed to `entmod`. There is one exception: `entmod` accepts new layer names.

If the entity list refers to a layer name that has not been defined in the current drawing, `entmod` creates a new layer. The attributes of the new layer are the standard default values used by the New option of the AutoCAD LAYER command.

The `entmod` function can modify subentities such as polyline vertices and block attributes.

If you use `entmod` to modify an entity in a block definition, this affects all INSERT or XREF references to that block. Also, entities in block definitions cannot be deleted by `entdel`.

## Adding an Entity to a Drawing

An application can add an entity to the drawing database by calling the `entmake` function. Like that of `entmod`, the argument to `entmake` is a list whose format is similar to that returned by `entget`. The new entity that the list describes is appended to the drawing database (it becomes the last entity in the drawing). If the entity is a complex entity (an old-style polyline or a block), it is not appended to the database until it is complete.

The following code fragment creates a circle on the MYLAYER layer:

```
(entmake ' ((0 . "CIRCLE")           ; Object type
            (8 . "MYLAYER")          ; Layer
            (10 5.0 7.0 0.0)         ; Center point
            (40 . 1.0)               ; Radius
          ) )
```

The following `entmake` restrictions apply to all entities:

- The first or second member in the list must specify the entity type. The type must be a valid DXF group code. If the first member does not specify the type, it can specify only the name of the entity: group -1 (the name is not saved in the database).
- AutoCAD must recognize all objects that the entity list refers to. There is one exception: `entmake` accepts new layer names.
- Any internal fields passed to `entmake` are ignored.
- `entmake` cannot create viewport entities.

For entity types introduced in AutoCAD Release 13 and later releases, you must also specify subclass markers (DXF group code 100) when creating the entity. All AutoCAD entities have the `AcDbEntity` subclass marker, and this must be explicitly included in the `entmake` list. In addition, one or more subclass marker entries are required to identify the specific sub-entity type.

These entries must follow group code 0 and must precede group codes that are specifically used to define entity properties in the entmake list. For example, the following is the minimum code required to entmake an MTEXT entity:

```
(entmake '(
  (0 . "MTEXT")
  (100 . "AcDbEntity")    ; Required for all post-R12 entities.
  (8 . "ALAYER")
  (100 . "AcDbMText")     ; Identifies the entity as MTEXT.
  (10 4.0 4.0 0.0)
  (1 . "Some\\Ptext")
)
```

The following table identifies the entities that do not require subentity marker entries in the list passed to entmake:

DXF names of entities introduced prior to AutoCAD Release 13	
3DFACE	ARC
ATTDEF	ATTRIB
CIRCLE	DIMENSION
INSERT	LINE
POINT	POLYLINE (old-style)
SEQEND	SHAPE
SOLID	TEXT
TRACE	VERTEX
VIEWPORT	

The entmake function verifies that a valid layer name, linetype name, and color are supplied. If a new layer name is introduced, entmake automatically creates the new layer. The entmake function also checks for block names, dimension style names, text style names, and shape names, if the entity type requires them. The function fails if it cannot create valid entities. Objects created on a frozen layer are not regenerated until the layer is thawed.

### Creating Complex Entities

To create a complex entity (an old-style polyline or a block), you make multiple calls to entmake, using a separate call for each subentity. When entmake



first receives an initial component for a complex entity, it creates a temporary file in which to gather the definition data and extended data, if present. (See “Extended Data—xdata” on page 323.) For each subsequent entmake call, the function checks if the temporary file exists. If it does, the new subentity is appended to the file. When the definition of the complex entity is complete (that is, when entmake receives an appropriate seqend or endblk subentity), the entity is checked for consistency; if valid, it is added to the drawing. The file is deleted when the complex entity is complete or when its creation has been canceled.

No portion of a complex entity is displayed on your drawing until its definition is complete. The entity does not appear in the drawing database until the final seqend or endblk subentity has been passed to entmake. The entlast function cannot retrieve the most recently created subentity for a complex entity that has not been completed. You can cancel the creation of a complex entity by entering entmake with no arguments. This clears the temporary file and returns nil.

As the previous paragraphs imply, entmake can construct only one complex entity at a time. If a complex entity is being created and entmake receives invalid data or an entity that is not an appropriate subentity, both the invalid entity and the entire complex entity are rejected. You can explicitly cancel the creation of a complex entity by calling entmake with no arguments.

The following example contains five entmake functions that create a single complex entity, an old-style polyline. The polyline has a linetype of DASHED and a color of BLUE. It has three vertices located at coordinates (1,1,0), (4,6,0), and (3,2,0). All other optional definition data assume default values. (For this example to work properly, the linetype DASHED must be loaded.)

```
(entmake ' ((0 . "POLYLINE")      ; Object type
            (62 . 5)              ; Color
            (6 . "dashed")        ; Linetype
            (66 . 1)              ; Vertices follow
          ) )
(entmake ' ((0 . "VERTEX")        ; Object type
            (10 1.0 1.0 0.0)      ; Start point
          ) )
(entmake ' ((0 . "VERTEX")        ; Object type
            (10 4.0 6.0 0.0)      ; Second point
          ) )
(entmake ' ((0 . "VERTEX")        ; Object type
            (10 3.0 2.0 0.0)      ; Third point
          ) )
(entmake ' ((0 . "SEQEND")))      ; Sequence end
```

When defining dotted pairs, as in the above example, there must be a space on both sides of the dot. Otherwise, you will get an invalid dotted pair error message.

Block definitions begin with a block entity and end with an `endblk` subentity. Newly created blocks are automatically entered into the symbol table where they can be referenced. Block definitions cannot be nested, nor can they reference themselves. A block definition can contain references to other block definitions.

---

**NOTE** Before you use `entmake` to create a block, you should use `tbl search` to ensure that the name of the new block is unique. The `entmake` function does not check for name conflicts in the block definitions table, so it can redefine existing blocks. See “Symbol Table and Dictionary Access” on page 332 for information on using `tbl search`.

---

Block references can include an attributes-follow flag (group 66). If present and equal to 1, a series of attribute (`attrib`) entities is expected to follow the insert object. The attribute sequence is terminated by a `seqend` subentity.

Old-style polyline entities always include a vertices-follow flag (also group 66). The value of this flag must be 1, and the flag must be followed by a sequence of vertex entities, terminated by a `seqend` subentity.

Applications can represent polygons with an arbitrarily large number of sides in polyface meshes. However, the AutoCAD entity structure imposes a limit on the number of vertices that a given face entity can specify. You can represent more complex polygons by dividing them into triangular wedges. AutoCAD represents triangular wedges as four-vertex faces where two adjacent vertices have the same value. Their edges should be made invisible to prevent visible artifacts of this subdivision from being drawn. The `PFACE` command performs this subdivision automatically, but when applications generate polyface meshes directly, the applications must do this themselves.

The number of vertices per face is the key parameter in this subdivision process. The `PFACEVMAX` system variable provides an application with the number of vertices per face entity. This value is read-only and is set to 4.

Complex entities can exist in either model space or paper space, but not both. If you have changed the current space by invoking either `MSPACE` or `PSPACE` (with `command`) while a complex entity is being constructed, a subsequent call to `entmake` cancels the complex entity. This can also occur if the subentity has a 67 group whose value does not match the 67 group of the entity header.

## Working with Blocks

There is no direct method for an application to check whether a block listed in the BLOCK table is actually referenced by an insert object in the drawing. You can use the following code to scan the drawing for instances of a block reference:

```
(ssget "x" ' ((2 . "BLOCKNAME"))) )
```

You must also scan each block definition for instances of nested blocks.

## Anonymous Blocks

The block definitions (BLOCK) table in a drawing can contain anonymous blocks (also known as unnamed blocks), which AutoCAD creates to support hatch patterns and associative dimensioning. The `entmake` function can create anonymous blocks other than `*Dnnn` (dimensions) and `*Xnnn` (hatch patterns). Unreferenced anonymous blocks are purged from the BLOCK definition table when a drawing is opened. Referenced anonymous blocks (those that have been inserted) are *not* purged. You can use `entmake` to create a block reference (insert object) to an anonymous block. (You *cannot* pass an anonymous block to the INSERT command.) Also, you can use `entmake` to redefine the block. You can modify the entities in a block (but not the block object itself) with `entmod`.

The name (group 2) of an anonymous block created by AutoLISP or ObjectARX has the form `*Unnn`, where `nnn` is a number generated by AutoCAD. Also, the low-order bit of an anonymous block's block type flag (group 70) is set to 1. When `entmake` creates a block whose name begins with `*` and whose anonymous bit is set, AutoCAD treats this as an anonymous block and assigns it a name. Any characters following the `*` in the name string passed to `entmake` are ignored.

---

**NOTE** Anonymous block names do not remain constant. Although a referenced anonymous block becomes permanent, the numeric portion of its name can change between drawing sessions.

---

## Entity Data Functions and the Graphics Screen

Changes to the drawing made by the entity data functions are reflected on the graphics screen, provided the entity being deleted, undeleted, modified, or made is in an area and on a layer that is currently visible. There is one exception: When `entmod` modifies a subentity, it does not update the image of the entire (complex) entity. If, for example, an application modifies 100 vertices of an old-style polyline with 100 calls to `entmod`, the time

required to recalculate and redisplay the entire polyline is unacceptably slow. Instead, an application can perform a series of subentity modifications, and then redisplay the entire entity with a single call to the `entupd` function.

Consider the following: If the first entity in the current drawing is an old-style polyline with several vertices, the following code modifies the second vertex of the polyline and regenerates its screen image.

```
(setq e1 (entnext))      ; Sets e1 to the polyline's entity name.
(setq v1 (entnext e1))    ; Sets v1 to its first vertex.
(setq v2 (entnext v1))    ; Sets v2 to its second vertex.
(setq v2d (entget v2))    ; Sets v2d to the vertex data.
(setq v2d
  (subst
    '(10 1.0 2.0 0.0)
    (assoc 10 v2d)        ; Changes the vertex's location in v2d
    v2d                  ; to point (1, 2, 0).
  )
)
(entmod v2d)             ; Moves the vertex in the drawing.
(entupd e1)              ; Regenerates the polyline entity e1.
```

The argument to `entupd` can specify either a main entity or a subentity. In either case, `entupd` regenerates the entire entity. Although its primary use is for complex entities, `entupd` can regenerate any entity in the current drawing.

---

**NOTE** To ensure that all instances of the block references are updated, you must regenerate the drawing by invoking the AutoCAD `REGEN` command (with `command`). The `entupd` function is not sufficient if the modified entity is in a block definition.

---

## Old-Style Polylines and Lightweight Polylines

A lightweight polyline (`lwpolyline`) is defined in the drawing database as a single graphic entity. The `lwpolyline` differs from the old-style polyline, which is defined as a group of subentities. `Lwpolylines` display faster and consume less disk space and RAM.

As of Release 14 of AutoCAD, 3D polylines are always created as old-style polyline entities, and 2D polylines are created as `lwpolyline` entities, unless they are curved or fitted with the `PEDIT` command. When a drawing from an earlier release is opened in Release 14 or a later release, all 2D polylines convert to `lwpolylines` automatically, unless they have been curved or fitted or contain `xdata`.

## Processing Curve-Fit and Spline-Fit Polylines

When an AutoLISP application uses `entnext` to step through the vertices of an old-style polyline, it might encounter vertices that were not created explicitly. Auxiliary vertices are inserted automatically by the PEDIT command's Fit and Spline options. You can safely ignore them, because changes to these vertices will be discarded the next time the user applies PEDIT to fit or to spline the polyline.

The old-style polyline entity's group 70 flags indicate whether the polyline has been curve-fit (bit value 2) or spline-fit (bit value 4). If neither bit is set, all the polyline's vertices are regular user-defined vertices. However, if the curve-fit bit (2) is set, alternating vertices of the polyline have the bit value 1 set in their 70 group to indicate that they were inserted by the curve-fitting process. If you use `entmod` to move the vertices of such a polyline with the intent of refitting the curve by means of PEDIT, ignore these vertices.

Likewise, if the old-style polyline entity's spline-fit flag bit (bit 4) is set, an assortment of vertices will be found—some with flag bit 1 (inserted by curve fitting if system variable `SPLINESEGS` was negative), some with bit value 8 (inserted by spline fitting), and all others with bit value 16 (spline frame-control point). Here again, if you use `entmod` to move the vertices and you intend to refit the spline afterward, move only the control-point vertices.

## Non-Graphic Object Handling

AutoCAD uses two types of non-graphic objects: dictionary objects and symbol table objects. Although there are similarities between these object types, they are handled differently.

All object types are supported by the `entget`, `entmod`, `entdel`, and `entmake` functions, although object types individually dictate their participation in these functions and may refuse any or all processing. With respect to AutoCAD built-in objects, the following rules apply. (All rules and restrictions that apply to graphic objects apply to non-graphic objects as well.) Non-graphic objects cannot be passed to the `entupd` function.

When using `entmake`, the object type determines where the object will reside. For example, if a layer object is passed to `entmake`, it automatically goes to the layer symbol table. If a graphic object is passed to `entmake`, it will reside in the current space (model or paper).

## Symbol Table Objects

The following rules apply to symbol tables:

- Symbol table entries can be created through `entmake` with few restrictions, other than being valid record representations, and name conflicts can only occur in the `VPORT` table. `*ACTIVE` entries cannot be created.
- Symbol table entries cannot be deleted with `entdel`.
- Symbol table and symbol table entry object states may be accessed with `entget` by passing the entity name. The `tbl obj name` function can be used to retrieve the entity name of a symbol table entry.
- Symbol tables themselves cannot be created with `entmake`, however, symbol table entries can be created with `entmake`.
- Handle groups (5, 105) may not be changed in `entmod`, nor specified in `entmake`.
- Symbol table entries that are not in the `APPID` table can have many of their fields modified with `entmod`. To be passed to `entmod`, a symbol table record list must include its entity name, which can be obtained from `entget` but not from the `tbl search` and `tbl next` functions. The 70 group of symbol table entries is ignored in `entmod` and `entmake` operations.

Renaming symbol table entries to duplicate names is not acceptable, except for the `VPORT` symbol table. The following entries may not be modified or renamed, except that most `LAYER` entries can be renamed and `xdata` can be modified on all symbol table entries.

### Symbol table entries that cannot be modified or renamed

Table	Entry name
VPORT	*ACTIVE
LINETYPE	CONTINUOUS
LAYER	Entries may not be modified, except for <code>xdata</code> , but renaming is allowed

The following entries may not be renamed but are otherwise modifiable, subject to restriction:

Symbol table entries that cannot be renamed	
Table	Entry name
STYLE	STANDARD
DIMSTYLE	STANDARD
BLOCKS	*MODEL_SPACE
BLOCKS	*PAPER_SPACE
APPID	No entries may be renamed

### Dictionary Objects

The following rules apply to dictionary objects:

- Dictionary objects can be examined with `entget` and their xdata modified with `entmod`. Their entries cannot be altered with `entmod`. All access to their entries are made through the `dictsearch` and `dictnext` functions.
- Dictionary entry contents cannot be modified through `entmod`, although xdata can be modified.
- Dictionary entries that begin with `ACAD*` cannot be renamed.

## Extended Data—xdata

Several AutoLISP functions are provided to handle extended data (xdata), which is created by applications written with ObjectARX or AutoLISP. If an entity contains xdata, it follows the entity's regular definition data.

You can retrieve an entity's extended data by calling `entget`. The `entget` function retrieves an entity's regular definition data and the xdata for those applications specified in the `entget` call.

When xdata is retrieved with `entget`, the beginning of extended data is indicated by a -3 code. The -3 code is in a list that precedes the first 1001 group.

The 1001 group contains the application name of the first application retrieved, as shown in the table and as described in the following sections.

Group codes for regular and extended data		
Group code	Field	Type of data
(-1, -2 (0-239 . . . )	Entity name) Regular definition data fields) . . .	Normal entity definition data
(-3 (1001 (1000, 1002-1071	Extended data sentinel Registered application name 1)  XDATA fields) . . .	Extended data
(1001 (1000, 1002-1071	Registered application name 2)  XDATA fields) . . .	
(1001	Registered application name 3) . .	

## Organization of Extended Data

Extended data consists of one or more 1001 groups, each of which begins with a unique application name. The xdata groups returned by `entget` follow the definition data in the order in which they are saved in the database.

Within each application's group, the contents, meaning, and organization of the data are defined by the application. AutoCAD maintains the information but does not use it. The table also shows that the group codes for xdata are in the range 1000-1071. Many of these group codes are for familiar data types, as follows:

String	1000. Strings in extended data can be up to 255 bytes long (with the 256th byte reserved for the null character).
Application Name	1001 (also a string value). Application names can be up to 31 bytes long (the 32nd byte is reserved for the null character) and must adhere to the rules for symbol table



names (such as layer names). An application name can contain letters, digits, and the special characters \$ (dollar sign), - (hyphen), and \_ (underscore). It cannot contain spaces.

Layer Name	1003. Name of a layer associated with the xdata.
Database Handle	1005. Handle of an entity in the drawing database.
3D Point	1010. Three real values, contained in a point.
Real	1040. A real value.
Integer	1070. A 16-bit integer (signed or unsigned).
Long	1071. A 32-bit signed (l ong) integer. If the value that appears in a 1071 group is a short integer or real value, it is converted to a l ong integer; if it is invalid (for example, a string), it is converted to a long zero (0L).

**NOTE** AutoLISP manages 1071 groups as real values. If you use `entget` to retrieve an entity's definition list that contains a 1071 group, the value is returned as a real, as shown in the following example:

```
(1071 . 12.0)
```

If you want to create a 1071 group in an entity with `entmake` or `entmod`, you can use either a real or an integer value, as shown in the following example:

```
(entmake ' ((. . . . (1071 . 12) . . . . )))
(entmake ' ((. . . . (1071 . 12.0) . . . . )))
(entmake ' ((. . . . (1071 . 65537.0) . . . . )))
(entmake ' ((. . . . (1071 . 65537) . . . . )))
```

But AutoLISP still returns the group value as a real:

```
(entmake ' ((. . . . (1071 . 65537) . . . . )))
```

The preceding statement returns the following:

```
(1071 . 65537.0)
```

ObjectARX always manages 1071 groups as long integers.

Several other extended data groups have special meanings in this context (if the application chooses to use them):

Control String	<p>1002. An xdata control string can be either "{ " or " } ". These braces enable the application to organize its data by subdividing it into lists. The left brace begins a list, and the right brace terminates the most recent list. Lists can be nested.</p> <p><b>NOTE</b> If a 1001 group appears within a list, it is treated as a string and does not begin a new application group.</p>
Binary Data	<p>1004. Binary data that is organized into variable-length chunks, which can be handled in ObjectARX with the <code>ads_binary</code> structure. The maximum length of each chunk is 127 bytes.</p> <p><b>NOTE</b> AutoLISP cannot directly handle binary chunks, so the same precautions that apply to Long (1071) groups apply to binary groups as well.</p>
World Space Position	<p>1011. Unlike a simple 3D point, the WCS coordinates are moved, scaled, rotated, and mirrored along with the parent entity to which the extended data belongs. The WCS position is also stretched when the STRETCH command is applied to the parent entity and when this point lies within the select window.</p>
World Space Displacement	<p>1012. A 3D point that is scaled, rotated, or mirrored along with the parent, but not stretched or moved.</p>
World Direction	<p>1013. A 3D point that is rotated or mirrored along with the parent, but not scaled, stretched, or moved. The WCS direction is a normalized displacement that always has a unit length.</p>
Distance	<p>1041. A real value that is scaled along with the parent entity.</p>
Scale Factor	<p>1042. Also a real value that is scaled along with the parent.</p>

The DXF group codes for xdata are also described in the *DXF Reference*.

## Registration of an Application

To be recognized by AutoCAD, an application must register the name or names that it uses. Application names are saved with the extended data of each entity that uses them, and also in the APPID table. Registration is done with the `regapp` function, which specifies a string to use as an application name. If it successfully adds the name to APPID, it returns the name of the application; otherwise it returns `nil`. A result of `nil` indicates that the name is already present in the symbol table. This is not an actual error condition but an expected return value, because the application name needs to be registered only once per drawing.

To register itself, an application should first check that its name is not already in the APPID table. If the name is not there, the application must register it. Otherwise, it can simply go ahead and use the data, as described later in this section.

The following fragment shows the typical use of `regapp`. (The `tbl search` function is described in “Symbol Table and Dictionary Access” on page 332.)

```
(setq appname "MYAPP_2356")      ; Unique application name.
(if (tbl search "appid" appname)  ; Checks if already registered.
    (princ (strcat
              "\n" appname " already registered. "))
    (if (= (regapp appname) nil)   ; Some other problem.
        (princ (strcat
                  "\nCan't register XDATA for " appname ". "))
    )
)
```

The `regapp` function provides a measure of security, but it cannot guarantee that two separate applications have not chosen the same name. One way of ensuring this is to adopt a naming scheme that uses the company or product name and a unique number (like your telephone number or the current date and time).

## Retrieval of Extended Data

An application can call `entget` to obtain the `xdata` that it has registered. The `entget` function can return both the definition data and the `xdata` for the applications it requests. It requires an additional argument, *application*, that specifies the application names. The names passed to `entget` must correspond to applications registered by a previous call to `regapp`; they can also contain wild-card characters.

By default, associative hatch patterns contain extended data. The following code shows the association list of this xdata.

Command: (entget (car (entsel)) '("ACAD"))

Select object: *Select an associative hatch*

Entering the preceding code at the command line returns a list that looks something like this:

```
((-1 . <Entity name: 600000c0>) (0 . "INSERT") (8 . "0") (2 . ""X0")
(10 0.0 0.0 0.0) (41 . 1.0) (42 . 1.0) (50 . 0.0) (43 . 1.0) (70 . 0) (71 . 0)
(44 . 0.0) (45 . 0.0) (210 0.0 0.0 1.0) (-3 ("ACAD" (1000 . "HATCH")
(1002 . "{") (1070 . 16) (1000 . "LINE") (1040 . 1.0) (1040 . 0.0)
(1002 . "}")"))))
```

This fragment shows a typical sequence for retrieving xdata for two specified applications. Note that the *appl i cat i on* argument passes application names in list form:

```
(setq working_elist
  (entget ent_name
    ' ("MY_APP_1" "SOME_OTHER") ; Only xdata from "MY_APP_1"
  ) ; and "SOME_OTHER" is retrieved.
)
(if working_elist
  (progn
    . . . ; Updates working entity groups.
    (entmod working_elist) ; Only xdata from registered
  ) ; applications still in the
) ; working_elist list are modified.
```

As the sample code shows, you can modify xdata retrieved by `entget` by using a subsequent call to `entmod`, just as you can use `entmod` to modify normal definition data. You can also create xdata by defining it in the entity list passed to `entmake`.

Returning the extended data of only those applications specifically requested protects one application from corrupting another application's data. It also controls the amount of memory that an application needs to use and simplifies the xdata processing that an application needs to perform.

---

**NOTE** Because the strings passed by *appl i cat i on* can include wild-card characters, an application name of "\*" will cause `entget` to return all extended data attached to an entity.

---

## Attachment of Extended Data to an Entity

You can use `xdata` to store any type of information you want. For example, draw an entity (such as a line or a circle), then enter the following code to attach `xdata` to the entity:

```
(setq lastent (entget (entlast))) ; Gets the association
                                   ; list of definition data
                                   ; for the last entity.
(regapp "NEWDATA")                ; Registers the
                                   ; application name.
(setq exdata                        ; Sets the variable
  '((-3 ("NEWDATA"                ; exdata equal to the
    (1000 . "This is a new thing!") ; new extended data-
  )))                             ; in this case, a text
                                   ; string.
)
(setq newent
  (append lastent exdata)) ; Appends new data list to
                           ; entity's list.
(entmod newent)           ; Modifies the entity with the new
                           ; definition data.
```

To verify that your new `xdata` has been attached to the entity, enter the following code and select the object:

```
(entget (car (entsel))) '("NEWDATA"))
```

This example shows the basic method for attaching extended data to an entity.

## Management of Extended Data Memory Use

Extended data is currently limited to 16K per entity. Because the `xdata` of an entity can be created and maintained by multiple applications, problems can result when the size of the `xdata` approaches its limit. AutoLISP provides two functions, `xdsize` and `xdroom`, to assist in managing the memory that `xdata` occupies. When `xdsize` is passed a list of `xdata`, it returns the amount of memory (in bytes) that the data will occupy. When `xdroom` is passed the name of an entity, it returns the remaining number of free bytes that can still be appended to the entity.

The `xdsize` function reads an extended data list, which can be large. This function can be slow, so it is not recommended that you call it frequently. A better approach is to use it (in conjunction with `xdroom`) in an error handler. If a call to `entmod` fails, you can use `xdsize` and `xdroom` to find out whether the call failed because the entity didn't have enough room for the `xdata`.

## Handles in Extended Data

Extended data can contain handles (group 1005) to save relational structures within a drawing. One entity can reference another by saving the other's handle in its xdata. The handle can be retrieved later from xdata and then passed to `handent` to obtain the other entity. Because more than one entity can reference another, xdata handles are not necessarily unique. The `AUDIT` command does require that handles in extended data either be `NULL` or valid entity handles (within the current drawing). The best way to ensure that xdata entity handles are valid is to obtain a referenced entity's handle directly from its definition data by means of `entget`. The handle value is in group 5.

When you reference entities in other drawings (for example, entities that are attached with `XREF`), you can avoid protests from `AUDIT` by using extended entity strings (group 1000) rather than handles (group 1005). The handles of cross-referenced entities are either not valid in the current drawing, or they conflict with valid handles. However, if an `XREF Attach` changes to an `XREF Bind` or is combined with the current drawing in some other way, it is up to the application to revise the entity references accordingly.

When drawings are combined by means of `INSERT`, `INSERT*`, `XREF Bind` (`XBIND`), or `partial DXFIN`, handles are translated so they become valid in the current drawing. (If the incoming drawing did not employ handles, new ones are assigned.) Extended entity handles that refer to incoming entities are also translated when these commands are invoked.

When an entity is placed in a block definition (with the `BLOCK` command), the entity within the block is assigned new handles. (If the original entity is restored by means of `OOPS`, it retains its original handles.) The value of any xdata handles remains unchanged. When a block is exploded (with the `EXPLODE` command), xdata handles are translated in a manner similar to the way they are translated when drawings are combined. If the xdata handle refers to an entity that is not within the block, it is unchanged. However, if the xdata handle refers to an entity that is within the block, the data handle is assigned the value of the new (exploded) entity's handle.

# Xrecord Objects

Xrecord objects are used to store and manage arbitrary data. They are composed of DXF group codes with normal object groups (that is, non-xdata group codes), ranging from 1 through 369 for supported ranges. This object is similar in concept to xdata but is not limited by size or order.

Xrecord objects are designed to work in such a way as not to offend releases R13c0 through R13c3. However, if read into a pre-R13c4 level of AutoCAD, xrecord objects disappear.

The following examples provide methods for creating and listing xrecord data.

```
(defun C:MAKEXRECORD( / xrec xname )
  ; create the xrecord's data list.
  (setq xrec ' ((0 . "XRECORD") (100 . "AcDbXrecord")
    (1 . "This is a test xrecord list")
    (10 1.0 2.0 0.0) (40 . 3.14159) (50 . 3.14159)
    (62 . 1) (70 . 180))
  )

  ; use entmakex to create the xrecord with no owner.
  (setq xname (entmakex xrec))

  ; add the new xrecord to the named object dictionary.
  (dictadd (namedobjdict) "XRECLIST" xname)

  (princ)
)

(defun C:LISTXRECORD ( / xlist )
  ; find the xrecord in the named object dictionary.
  (setq xlist (dictsearch (namedobjdict) "XRECLIST"))

  ; print out the xrecord's data list.
  (princ xlist)

  (princ)
)
```

# Symbol Table and Dictionary Access

AutoLISP provides functions for accessing symbol table and dictionary entries. Examples of the `tbl next` and `tbl search` functions are provided in the following sections. For a complete list of the symbol table and dictionary access functions, see “Symbol Table and Dictionary-Handling Functions” in appendix A, “AutoLISP Function Synopsis.” Refer to the *AutoLISP Reference* for more detailed information on the functions listed in the Synopsis.

For additional information on non-graphic objects see, “Non-Graphic Object Handling” on page 321.

## Symbol Tables

Symbol table entries can also be manipulated by the following functions:

- `entdel`
- `entget`
- `entmake`
- `entmod`
- `handent`

The `tbl next` function sequentially scans symbol table entries, and the `tbl search` function retrieves specific entries. Table names are specified by strings. The valid names are `LAYER`, `LTYPE`, `VIEW`, `STYLE`, `BLOCK`, `UCS`, `VPORT`, `DIMSTYLE`, and `APPID`. Both functions return lists with DXF group codes that are similar to the entity data returned by `entget`.

The first call to `tbl next` returns the first entry in the specified table. Subsequent calls that specify the same table return successive entries, unless the second argument to `tbl next` (*rewind*) is nonzero, in which case `tbl next` returns the first entry again.

In the following example, the function `GETBLOCK` retrieves the symbol table entry for the first block (if any) in the current drawing, and then displays it in a list format.

```
(defun C:GETBLOCK (/ blk ct)
  (setq blk (tbl next "BLOCK" 1)) ; Gets the first BLOCK entry.
  (setq ct 0) ; Sets ct (a counter) to 0.
  (textpage) ; Switches to the text screen.
  (prin1 "\nResults from GETBLOCK: ")
```



```

(repeat (length blk)          ; Repeats for the number of
                                ; members in the list.
  (print (nth ct blk))        ; Prints a new line, then
                                ; each list member.
  (setq ct (1+ ct))          ; Increments the counter by 1.
)
(print)                       ; Exits quietly.
)

```

Entries retrieved from the BLOCK table contain a -2 group that contains the name of the first entity in the block definition. If the block is empty, this is the name of the block's ENDBLK entity, which is never seen on occupied blocks. In a drawing with a single block named BOX, a call to GETBLOCK displays the following. (The name value varies from session to session.)

Results from GETBLOCK:

```

(0 . "BLOCK")
(2 . "BOX")
(70 . 0)
(10 9.0 2.0 0.0)
(-2 . <Entity name: 40000126>)

```

As with `tbl next`, the first argument to `tbl search` is a string that names a table, but the second argument is a string that names a particular symbol in the table. If the symbol is found, `tbl search` returns its data. This function has a third argument, *setnext*, that you can use to coordinate operations with `tbl next`. If *setnext* is `nil`, the `tbl search` call has no effect on `tbl next`, but if *setnext* is `non-nil`, the next call to `tbl next` returns the table entry following the entry found by `tbl search`.

The *setnext* option is useful when you are handling the VPORT symbol table, because all viewports in a particular viewport configuration have the same name (such as `*ACTIVE`).

If the VPORT symbol table is accessed when TILEMODE is turned off, any changes have no visible effect until TILEMODE is turned on. Do not confuse VPORTS, which is described by the VPORT symbol table with paper space viewport entities.

The following processes all viewports in the 4VIEW configuration:

```

(setq v (tbl search "VPORT" "4VIEW" T)) ; Finds first VPORT entry.
(while (and v (= (cdr (assoc 2 v)) "4VIEW"))
  .
                                ; ... Processes entry ...
  .
  (setq v (tbl next "VPORT"))      ; Gets next VPORT entry.
)

```

## Dictionary Entries

A dictionary is a container object, similar to the symbol tables in functions. Dictionary entries can be queried with the `dictsearch` and `dictnext` functions. Each dictionary entry consists of a text name key plus a hard ownership handle reference to the entry object. Dictionary entries may be removed by directly passing entry object names to the `entdel` function. The text name key uses the same syntax and valid characters as symbol table names.

### Accessing AutoCAD Groups

The following is an example of one method for accessing the entities contained in a group. This example assumes a group named `G1` exists in the current drawing.

```
(setq objdict (namedobjdict))  
(setq grpdict (dictsearch objdict "ACAD_GROUP"))
```

This sets the `grpdict` variable to the entity definition list of the `ACAD_GROUP` dictionary and returns the following:

```
((-1 . <Entity name: 8dc10468>) (0 . "DICTIONARY") (5 . "D")  
(102 . "{ACAD_REACTORS}") (330 . <Entity name: 8dc10460>)  
(102 . "}") (100 . "AcDbDictionary") (3 . "G1")  
(350 . <Entity name: 8dc41240>))
```

The following code sets the variable `group1` to the entity definition list of the `G1` group:

```
(setq group1 (dictsearch (cdar grpdict) "G1"))
```

It returns the following:

```
((-1 . <Entity name: 8dc10518>) (0 . "GROUP") (5 . "23")  
(102 . "{ACAD_REACTORS}") (330 . <Entity name: 8dc10468>)  
(102 . "}") (100 . "AcDbGroup") (300 . "line and circle") (70 . 0) (71 . 1)  
(340 . <Entity name: 8dc10510>)(340 . <Entity name: 8dc10550> )
```

The 340 group codes are the entities that belong to the group.

# Part 3

## Working with Programmable Dialog Boxes





# Designing Dialog Boxes

Dialog boxes are defined by ASCII files written in *dialog control language* (DCL). The elements in a dialog box, such as buttons and edit boxes, are known as tiles. The size and functionality of each tile is controlled by the tile's attributes. The size of the dialog box and the layout of its parts are set automatically with a minimum of positioning information. VLISP provides a tool for viewing dialog boxes, and provides functions for controlling dialog boxes from application programs.

This chapter introduces the elements that make up dialog boxes. It explains DCL file structure and syntax, and presents sample AutoLISP and DCL code for a sample dialog box. This chapter also provides some DCL coding techniques for handling layout problems.

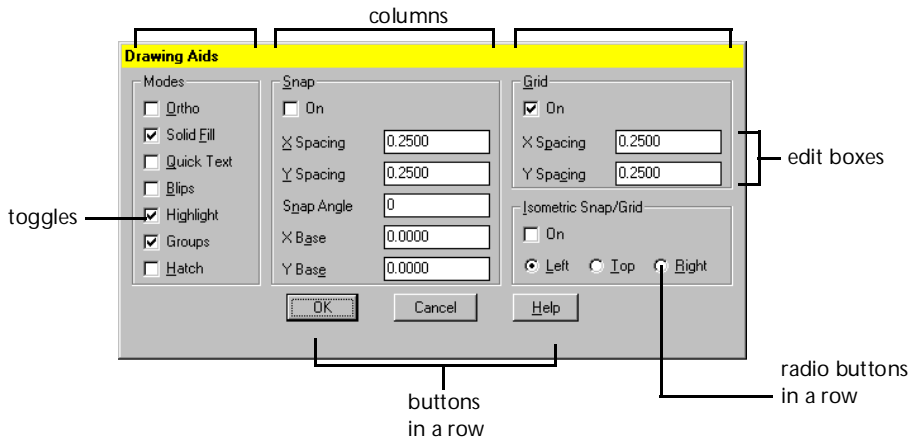
## In This Chapter

# 11

- Dialog Box Components
- Using DCL to Define Dialog Boxes
- Displaying Dialog Boxes with Visual LISP
- Adjusting the Layout of Dialog Boxes
- Design Guidelines

# Dialog Box Components

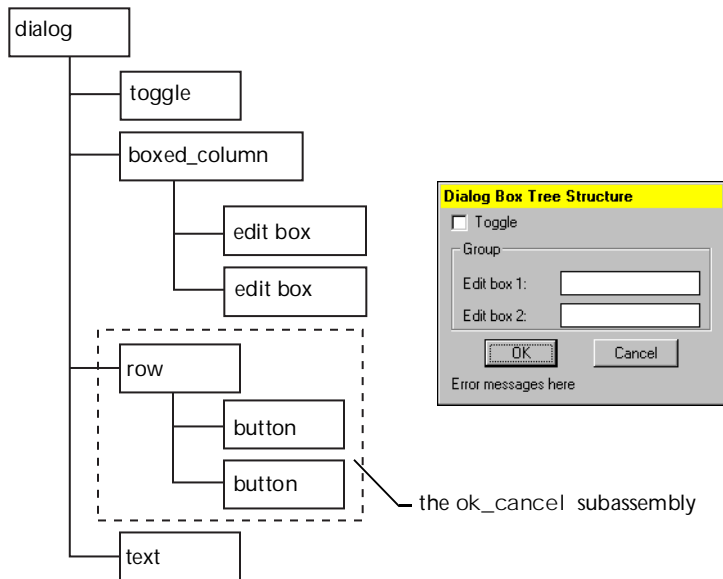
The following figure shows a standard AutoCAD dialog box, with some of its components labeled. In dialog box creation and customization these components are known as tiles.



A dialog box consists of the box and the tiles within it. The basic tile types are predefined by the programmable dialog box (PDB) facility.

You can create complex tiles, called subassemblies, by grouping tiles into rows and columns, with or without an enclosing box or border. A row or column of tiles is referred to as a cluster. Subassemblies define groups of tiles or clusters that can be used in many dialog boxes. For example, the OK, Cancel, and Help buttons are grouped into a subassembly, defined as a row (cluster) of three button tiles and some spacing separating the buttons.

Subassemblies are treated as single tiles. The tiles within a subassembly are called children. DCL files are organized in a tree structure. At the top of the tree is a (di al og) tile that defines the dialog box itself. The following diagram shows a DCL file structure:



The layout, appearance, and behavior of a tile or subassembly are specified in DCL by the tile's attributes. For example, the `dialog` itself, and most predefined tile types, has a `label` attribute that specifies the text associated with the tile. The `label` of a dialog box defines the caption at the top of the dialog box, the `label` of a button specifies the text inside the button, and so on.

DCL also enables you to define new tiles, called prototypes, that are not necessarily associated with a specific dialog box. This is useful when you want to use the same component in several dialog boxes. You can reference prototype tiles from other DCL files and change their attributes the same way you change predefined tiles.

Before you program a dialog box, plan both the dialog box and the application in detail before you code and debug. The sequence in which the data is entered will vary with each user. The need to anticipate a variety of user actions imposes a program structure that is less linear than conventional programming, but is more reflective of the way users work.

# Using DCL to Define Dialog Boxes

You define dialog boxes by entering DCL descriptions in ASCII text files, much like writing AutoLISP code. DCL files have a *.dcl* extension. A single DCL file can contain the description of one or more dialog boxes, or it can contain only prototype tiles and subassemblies for use by other DCL files. A DCL file consists of the following three parts, which can appear in any order. Depending on your application, only one or more of these parts is required.

- References to other DCL files

These consist of include directives as described in “Referencing DCL Files” on page 340.

- Prototype tile and subassembly definitions

These are tile definitions you can refer to in subsequent tile definitions (including dialog box definitions).

- Dialog box definitions

These define the attributes of tiles or override the attributes defined in prototype tiles and subassemblies.

## The *base.dcl* and *acad.dcl* Files

The *base.dcl* and *acad.dcl* files are included with AutoCAD and are distributed in the AutoCAD Support directory.

The *base.dcl* file shows the DCL definitions for the basic, predefined tiles and tile types. It also contains definitions for commonly used prototypes. The PDB feature does not allow you to redefine the predefined tiles. The *acad.dcl* file contains the standard definitions of all the dialog boxes used by AutoCAD.

---

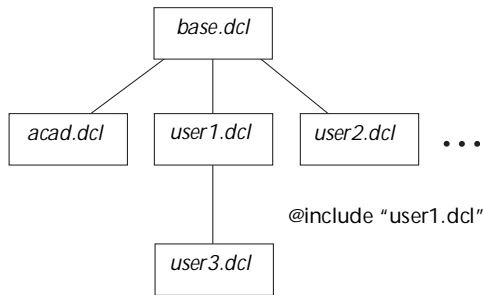
**WARNING!** Any errors in *base.dcl* may disrupt the appearance of the standard AutoCAD dialog boxes and customized dialog boxes from your application or other applications.

---

## Referencing DCL Files

When you create dialog boxes, you must create a new, application-specific DCL file. All DCL files can use the tiles defined in the *base.dcl* file. A DCL file can also use tiles defined in another DCL file by naming the other file in what is called an include directive. You can create your own hierarchy of DCL files, as shown in the following figure:





In this figure, the *user1.dcl* and *user2.dcl* files are independent of each other, but *user3.dcl* uses tiles defined in *user1.dcl*. The include directive has the form:

```
@include filename
```

where *filename* is a quoted string containing the full name of the other DCL file. For example, the following directive includes a file named *usercore.dcl*:

```
@include "usercore.dcl"
```

If you specify only the file name, the PDB feature searches for the file first in the current directory and then in the same directory as the DCL file itself (the one that contains the include directive). If you specify a full path name, the PDB feature searches only the directory specified in that path.

---

**NOTE** The DCL files you create cannot use the dialog boxes defined in *acad.dcl*. You cannot specify `@include "acad.dcl"`. However, if you want to create similar dialog boxes, you can cut and paste the definitions into your own DCL file.

---

## DCL Syntax

This section describes the DCL syntax for specifying tiles, tile attributes, and attribute values.

New tiles are created by tile definitions. If a tile definition appears outside a dialog box definition, it is a prototype or a subassembly. Prototypes and subassemblies can be used in dialog box definitions by tile references. Each reference to a definition inherits the attributes of the original tile. When referring to prototypes, you can change the values of the inherited attributes or add new attributes. When referring to subassemblies, you cannot change or add attributes.

If you need multiple instances of a tile with some attributes in common, it is easiest to define and name a prototype that contains only the common

attributes. Then, in each reference to the prototype, you can change attributes or add new ones, but you do not have to list all the common attributes each time you reference the tile. Because attributes are inherited, you will more often need to create tile references—especially references to the predefined tiles—than to define new tiles.

## Tile Definitions

Tile definitions have the following form:

```
name : item1 [ : item2 : item3 ... ] {  
    attribute = value;  
    ...  
}
```

where each *item* is a previously defined tile. The new tile (*name*) inherits the attributes of all the specified tiles (*item1*, *item2*, *item3*, ...). The attribute definitions within the curly braces ({}) either supplement, or, if the attribute's name is identical, replace the inherited definitions. When the definition has multiple parents, attributes take precedence in left-to-right order. In other words, if more than one item specifies the same attribute, the first one encountered is the one used.

If the new definition contains no children, it is a prototype, and you can alter or augment its attributes when referring to it. If it is a subassembly with children, its attributes cannot be altered.

The *name* of a tile or tile prototype can contain only letters, numbers, or the underscore character (\_), and must begin with a letter.

---

**NOTE** Tile names are case-sensitive. For example, `bigbutton` is not the same as `BigButton` or `BIGBUTTON`. Be careful when using capitalization.

---

This is the (internal) definition of a button:

```
button : tile {  
    fixed_height = true;  
    is_tab_stop = true;  
}
```

The *base.dcl* file defines a default `t_button` as follows:

```
default t_button : button {  
    is_default = true;  
}
```

The `default t_button` inherits the button tile's values for the `fixed_height` and `is_tab_stop` attributes. It adds a new attribute, `is_default`, and sets it to `true`.

## Tile References

Tile references have one of the following forms:

```
name;
```

or

```
: name {  
    attribute = value;  
    . . .  
}
```

where *name* is the name of a previously defined tile. Tile names are case sensitive. In the first instance, all the attributes defined in *name* are incorporated into the reference. In the second instance, the attribute definitions within the curly braces either supplement or replace the definitions inherited from *name*. Because this is a tile reference, as opposed to a definition, the attribute changes apply only to this instance of the tile.

---

**NOTE** The format of the second instance can refer only to prototypes, not to subassemblies.

---

The `spacer` tile is used for layout in a dialog box definition. It has no unique attributes, so references to it specify only its name:

```
spacer;
```

The `ok_cancel` tile defined in *base.dcl* is a subassembly, so it too can be referenced only by name:

```
ok_cancel ;
```

On the other hand, you have the option of redefining the attributes of an individual tile. For example, the following statement creates a button with the same properties as a previously defined button, but with different text:

```
: retirement_button {  
    label = "Goodbye";  
}
```

For more information, see “Customizing Exit Button Text” on page 351.

## Attributes and Attribute Values

Within the curly braces of a tile definition or reference, you specify attributes and assign them values using the following form:

```
attribute = value ;
```

where *attribute* is a valid keyword and *value* is the value assigned to the attribute. An equal sign (=) separates the attribute from the value, and a semi-

colon (:) ends the assignment statement. For example, the key attribute defines the name by which a program can refer to the tile, and the label attribute defines the text displayed within the tile. See “Synopsis of Pre-defined Attributes” on page 394 for a complete list of DCL attributes.

As with tile names, attribute names and values are also case-sensitive. `Width` and `width` are not the same thing; `True` and `true` do not produce the same results.

## Comments

A statement preceded by two forward slashes (//) is treated as a comment in a DCL file. Anything that appears between the // and the end of the line is ignored. DCL also allows C language style comments. These have the form `/* comment text */`. The starting `/*` and ending `*/` can be on separate lines.

# Displaying Dialog Boxes with Visual LISP

VLISP provides a tool for previewing dialog boxes defined with DCL. To see how this works, copy the following DCL code into a new file in the VLISP text editor:

```
hello : dialog {
  label = "Sample Dialog Box";
  : text {
    label = "Hello, world";
  }
  : button {
    key = "accept";
    label = "OK";
    is_default = true;
  }
}
```

This DCL file defines a dialog box labeled “Sample Dialog Box.” It contains a text tile and an OK button. Save the file as *hello.dcl*, and specify “DCL Source Files” in the Save As Type field of the Save As dialog box.

---

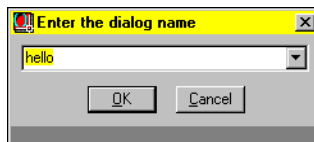
**NOTE** You must include the file extension when you specify the file name. VLISP does not automatically add a *.dcl* file extension for you.

---

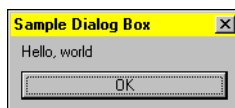
Note how the text editor color codes the statements in the DCL file. The default color coding scheme is shown in the following table:

DCL default color coding	
DCL element	Color
Tiles and tile attributes	Blue
Strings	Magenta
Integers	Green
Real numbers	Teal
Comments	Magenta, on gray background
Parentheses	Red
Preprocessor	Dark blue
Operators and punctuation	Dark red
Unrecognized items (for example, user variables)	Black

Choose **Tools** ► **Interface Tools** ► **Preview DCL in Editor** to display the dialog box defined in the text editor window. Because you may have more than one dialog box defined in a single *.dcl* file, VLISP prompts you to specify the name of the dialog you want to view:



If your DCL file contains definitions for multiple dialog boxes, click the pull-down arrow and choose the one you want to preview. There is only one dialog box defined in *hello.dcl*, so choose OK to view it:



Choose OK to complete previewing the dialog box.

Although buttons are a good way to demonstrate dialog box attributes, there are standard exit button subassemblies you should use in your dialog boxes. You can create a dialog box that is virtually the same as the one shown in the previous figure by using the `ok_only` subassembly. See “Dialog Box Exit Buttons and Error Tiles” on page 410.

## Preview Error Handling

If your DCL code contains errors, the VLISP DCL previewer displays messages indicating the offending line and keyword or symbol. For example, introduce an error into *hello.dcl* by removing the colon before “button,” then try previewing the dialog box. You’ll see the following message:



Press OK to clear the message from your screen. VLISP may display additional error messages, like the following:



## Semantic Auditing of DCL Files

AutoCAD provides a choice of four levels (0–3) of semantic auditing for DCL files (see the following table). Auditing attempts to detect code in the DCL file that is likely to be problematic or unnecessary. These audits are done at DCL load time. To set the audit level for a DCL file, include a line such as the following anywhere within the DCL file, but not inside any tile definitions:

```
dcl_settings : default t_dcl_settings { audit_level = 3; }
```

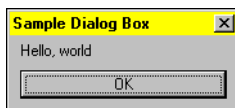
If your DCL file references other DCL files with include directives, you should define `dcl _settings` in only one file. The defined audit level is used in all included files. The following table describes each audit level:

Semantic auditing levels	
Level	Description
0	No checking. Use only if the DCL files have been audited and have not been touched since the audit.
1	Errors. Finds DCL bugs that may cause AutoCAD to terminate. This level of checking is the default and involves almost no delay. Errors can include using undefined tiles and circular prototype definitions.
2	Warnings. Finds DCL bugs that result in dialog boxes with undesired layout or behavior. A modified DCL file should be audited at this level at least once. The warning level catches mistakes such as missing required attributes and inappropriate attribute values.
3	Hints. Finds redundant attribute definitions.

To get the most out of the auditing facility, you should keep the `audit_level` at 3 during program development. Remember to strip out the `dcl _settings` line before shipping DCL files to users.

## Adjusting the Layout of Dialog Boxes

Look again at the sample dialog box defined in the previous section. There is a small problem with it:



Notice how the OK button occupies almost the full width of the dialog box. To improve the appearance of this dialog box, you can edit the DCL file and add two attributes to the button tile. To prevent the button from filling the available space, add a `fixed_width` attribute and set it to true. This causes the button's border to shrink so that it is just slightly wider than the text inside.

To center the button, add an alignment attribute that specifies centered. Tiles in a column are left-justified by default. Now the DCL description is as follows:

```
hello : dialog {
    label = "Sample Dialog Box";
    : text {
        label = "Hello, world";
    }
    : button {
        key = "accept";
        label = "OK";
        is_default = true;
        fixed_width = true;
        alignment = centered;
    }
}
```

The dialog box now appears like the following:



Many common layout problems can be resolved with the techniques that are described in the following subsections. If the default layout is not suitable to the dialog box you are creating, adjust the layout by changing the defaults at the prototype or subassembly level. Adjust individual tiles only when necessary.

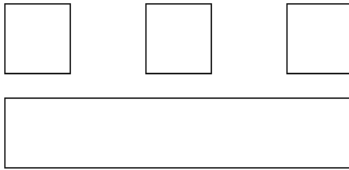
## Distributing Tiles in a Cluster

When laying out tiles in a dialog box, you need to arrange them into rows and columns based on the relative size of each tile. The following DCL defines a row of three tiles that runs along the top of another tile:

```
: column {
    : row {
        : compact_tile {
        }
        : compact_tile {
        }
        : compact_tile {
        }
    }
    : large_tile {
    }
}
```



If the `compact_tile` components have `fixed_width` and the `large_tile` is wider than the minimum space required by the row of `compact_tiles` above it, the default horizontal alignment of this assembly appears as follows:

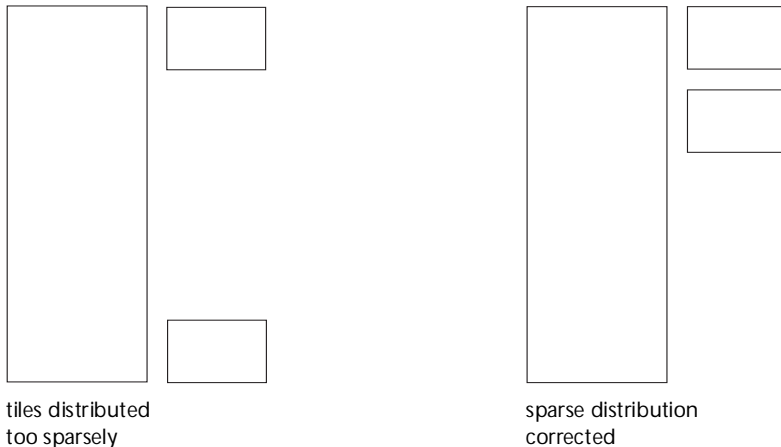


The leading edge of the first `compact_tile` in the row aligns with the leading edge of the `large_tile`, and the trailing edge of the last `compact_tile` aligns with the trailing edge of the `large_tile`. Tiles in between are distributed evenly. The situation with adjoining columns is analogous.

You can control the default distribution by using the `spacer_0` and `spacer_1` tiles, which are variants of the `spacer` tile defined in *base.dcl*. See “DCL Tile Catalog” on page 410 for more information on these tiles.

## Adjusting the Space between Tiles

If two adjoining columns differ greatly in the amount of space their tiles occupy, then the tiles in the one that needs less space may appear to be distributed too far apart. Their appearance can be improved if you set the incongruous column’s `fixed_height` attribute to `true`. The result for vertical tile distribution is shown in the following diagram:



## Adjusting Space at the Right Side or Bottom

A dialog box may contain unused space along its right side. You can define a text tile and explicitly specify a width greater than the width required by its current value. For example, the following code fragment defines a tile that does not display anything (its value is null) until an application sets its value:

```
: text {  
    key = "l_text";  
    width = 18;  
    fixed_width = true;  
}
```

The `width` attribute reserves space for 18 characters in the dialog box. The application can add text with a statement like the following:

```
(set_tile "l_text" "By layer")
```

Because "By layer" doesn't need all 18 characters, the dialog box has surplus space along its right side.

A similar situation occurs when you use an `errtile` to display error messages. (See "Dialog Box Exit Buttons and Error Tiles" on page 410.) Unless an error message is currently shown, it looks as if there is extra space at the bottom of the dialog box. In this case, an extra spacer tile at the top of the dialog box can help balance the vertical layout.

## Fixing the Spacing Around a Boxed Row or Column

If the `label` attribute of a boxed row or column is either blank (" ") or null (""), the box encloses the cluster but no text is shown. A single blank does not appear as a space in the box. However, there is a difference in the way blank and null labels are laid out:

- If the `label` is a single blank, any vertical space the text occupied inside the box is lost, but any vertical space the label occupied above the box is not lost.
- If the label is a null string, all vertical space is lost, whether above the box or inside it.

In the following DCL code, the top lines of the boxes around the first two columns are guaranteed to line up (with the same `Ylocation`), and the top line of the box around the third column is guaranteed to have no spacing above or below it, except for the default margins:

```

: row {
    : boxed_col umn {
        label = "Some Text";
    }
    : boxed_col umn {
        label = " ";           // single blank: the default
    }
    : boxed_col umn {
        label = "";           // null string
    }
}

```

## Customizing Exit Button Text

For some dialog boxes, you may want to change the text of one of the exit buttons. For example, if you create a dialog box capable of destroying data, it's safer to call the button Destroy instead of OK. To do this, use the retirement\_button prototype as follows:

```

destroy_button : retirement_button {
    label = "&Destroy";
    key = "destroy";
}

```

Notice the use of the ampersand (&) in the label attribute. This assigns a mnemonic to the tile. In this case the letter D is underscored in the button label and becomes the mnemonic.

---

**NOTE** When customizing existing button subassemblies you should obtain the proper DCL code from your *base.dcl* file rather than from the manual.

---

Once you have defined a custom exit button, you need to embed it in a subassembly that matches the appearance and functionality of the standard clusters. The following example shows the current definition of ok\_cancel\_help:

```

ok_cancel_help : column {
    : row {
        fixed_width = true;
        alignment = centered;
        ok_button;
        : spacer { width = 2; }
        cancel_button;
        : spacer { width = 2; }
        help_button;
    }
}

```

Create a new subassembly that replaces the `ok_button` with the new button as follows:

```
destroy_cancel_help : column {  
    : row {  
        fixed_width = true;  
        alignment = centered;  
        destroy_button;  
        : spacer { width = 2; }  
        cancel_button;  
        : spacer { width = 2; }  
        help_button;  
    }  
}
```

In the standard subassembly, the OK button is the default, but this attribute wasn't added to `destroy_button`. Where the dialog box's action can be destructive (or very time-consuming), it is strongly recommended to make the Cancel button the default. In this case, it functions both as the default and as the Abort button:

```
destroy_cancel_help : column {  
    : row {  
        fixed_width = true;  
        alignment = centered;  
        destroy_button;  
        : spacer { width = 2; }  
        : cancel_button { is_default = true; }  
        : spacer { width = 2; }  
        help_button;  
    }  
}
```

Because an attribute has been changed, the original Cancel button is used as a prototype, requiring a colon in front of `cancel_button`.

---

**WARNING!** When the Cancel button and the Default button are the same (both `is_default` and `is_cancel` are true) and you neglect to assign an action that calls `done_dialog` to any other button, then no other button can exit the dialog box and it will always be canceled.

---

# Design Guidelines

To design a dialog box well, you must consider not only the practical purpose of the box but also its aesthetics, the ergonomics of using it, and the GUI standards for the Windows environment. The following subsections provide some guidelines for GUI design, dialog box design, and predefined tiles and clusters. Refer to the “Programmable Dialog Box Reference,” chapter 13, for more examples of tiles and clusters.

## Aesthetics and Ergonomics

The appearance of a dialog box is important. If the box is too cluttered, it is ineffective and hard to use. Also, tiles should be arranged so they are easy to use. Consider which tiles will be used most frequently, make them prominent in the design, and arrange them so it's easy to move between them, particularly when they are used in conjunction with each other.

## Consistent Design and Clear Language

The user interface of an application should be internally consistent, and consistent with related applications. An unfamiliar dialog box is easier to understand if its design is consistent with other dialog boxes in the application, related applications, or the host system. One example of this is the consistent placement of buttons such as OK and Cancel. The technique associated with each kind of tile—how you enter text in a text box and how you select a list box item—should also be consistent. The best way to achieve consistency is to reuse tiles and the code that controls them.

Standardization contributes to consistency. Use standard definitions for dialog box controls. This reduces your work, contributes to consistency, and makes it easier for users to learn and use your dialog boxes.

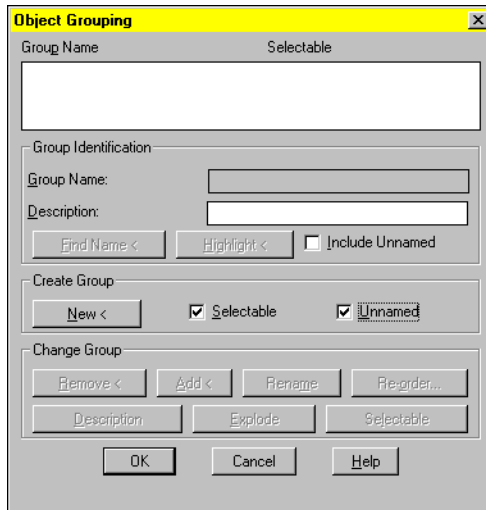
Use language that is clear. Although dialog boxes are considered part of a graphical interface, most of the tiles and information they present are textual. The labeling of dialog boxes, naming of buttons, and phrasing of messages should be direct and unambiguous. Avoid jargon and technical terms that users may not understand.

## User Control

Give users some control over how they access the dialog box to enter input. One advantage of using dialog boxes instead of a command line interface is that boxes don't confine users to a strict sequence of prompts. In a dialog

box, users should be able to enter input in any sequence. Some constraints are necessary—when selecting one option causes another to be unavailable, for example—but build in only constraints that have underlying reasons in the way your application works.

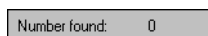
For example, the following figure shows the Object Grouping dialog box. This dialog box contains a Group Name field, where users may enter a name for a new group they are creating. If the Unnamed option is selected, a Group Name cannot be specified.



Multiple dialog boxes should appear on top of one another rather than require the user to exit the current box before calling another. Always let users return to the dialog box that was initially displayed. This design doesn't commit users to a choice before they are ready to leave the dialog box. Because the current dialog box appears on top of the previous one, it reminds users of the context: where they have come from and where they'll return to.

Whenever users do something to change the current status or options, provide them with immediate feedback. If users select something, show it or describe it immediately. If one choice excludes other choices, be sure to make the invalid choices unavailable immediately.

In the AutoCAD Color Selection dialog box, for example, an image tile shows the color immediately after the user selects its number. In the sample Block Definition dialog box, the number of selected objects is always displayed in a message below the Select Objects button:

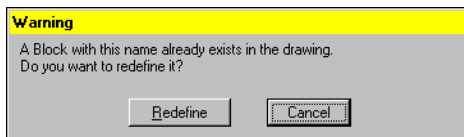


## Forgiving Errors

Make your dialog boxes forgiving, so users feel free to explore without fear of making irreversible mistakes. Report minor errors by messages in an error tile at the bottom of the dialog box. Report more serious errors by displaying an alert box. The `alert` function displays a simple alert box (with a single OK button). See “`alert`” in the *AutoLISP Reference*.

If the user selects a potentially destructive or time-consuming action, the dialog box should display an alert box that gives the user a choice of proceeding with the operation or canceling it.

For example, in the Block Definition dialog box, an alert box appears when users attempt to create a block that already exists. Users can then choose to proceed and overwrite the original block, or cancel the operation without making changes:



Nested dialog boxes that alert users should return to the previous dialog box. Terminate the current nest of dialog boxes only in the case of serious or potentially fatal errors.

## Providing Help

You should provide a Help facility. How much online help you provide depends on how complex your application is and how self-explanatory your dialog boxes are. At the very least, it is recommended that the main dialog box of your application have a Help button that displays a single dialog box describing important information. In most cases, the Help button should call the Help facility using the `help` function.

If your application is more sophisticated, consider developing a context-sensitive Help facility with multiple Help dialog boxes, each associated with a particular dialog box.

## Users with Disabilities

Considerations intended for users with disabilities can make a program easier for anyone to use. When designing your dialog boxes, consider the following:

Color	<p>Many people cannot distinguish between certain colors. If you use color coding to present information, supplement this by presenting the same information in some other way (usually with text).</p> <p>For example, the standard AutoCAD color dialog box displays a text message that states the color's name or number as well as an image tile that displays the color.</p>
Online Help	<p>Many users have difficulty either reading the small print in manuals or physically handling books. Even a single Help button on the main dialog box can be useful.</p>
Keyboard Access	<p>Some users may have difficulty with or be unable to use a pointing device. Try to specify mnemonics so your dialog boxes can be used with just the keyboard.</p>
Clarity and Simplicity	<p>Carefully designed dialog boxes with clear and simple language help users with verbal or cognitive impairments. Don't force users to remember many different things. Instead, use consistent terminology and present choices wherever possible.</p>

## Using Capitalization

The following are some general guidelines for capitalizing text within dialog boxes:

Dialog Boxes, Areas, and Column Headings	<p>Use headline capitalization: capitalize the first and last words, and all other words except articles, prepositions, and coordinating conjunctions. However, if the dialog box is invoked from a menu (not from the Command prompt), its title should match the menu item.</p>
Control Labels	<p>Use headline capitalization for labels of control tiles such as buttons. Do not follow labels with a period. Follow the labels of a text box or a drop-down list with a colon (:). You may want to use sentence-style capitalization (in</p>



which you capitalize only the first word and proper nouns) if the label is lengthy or phrased as a question.

Prompts and  
Messages

Use sentence-style capitalization.

## Avoiding Abbreviations

Abbreviations can be ambiguous and difficult to translate. If space constraints require you to abbreviate terms, abbreviate them consistently within a group (such as a boxed column). Be consistent. Don't spell some terms in full and abbreviate others.

## Layout

Arrange sections of the dialog box logically into rows or columns so users can scan them from left to right or from top to bottom. Align related entry fields (such as edit boxes or list boxes) both vertically and horizontally, so that when users switch fields by pressing TAB, the cursor moves in a straight, orthogonal line.

If there is a natural order for entering data—such as the *X*, *Y*, and *Z* of coordinates—order the fields in the same way. Align boxed areas both vertically and horizontally. Do not leave a lot of white space around or between boxed areas. Extend their width to the right, if necessary.

## Size and Placement

To display information clearly, make the dialog box no larger than necessary.

---

**NOTE** Users may have a screen resolution as low as 640 × 480. If you are developing applications on displays using a higher resolution, remember to verify that your dialog boxes display properly at lower resolutions.

---

By default, AutoCAD initially displays all dialog boxes in the center of the graphics window. However, you can specify that dialog boxes display at an alternate location (such as the last location specified by the user). The `new_dialog` and `done_dialog` AutoLISP functions provide for dialog box placement.

## Disabling Tiles

If a tile or an area is unavailable or irrelevant given the current option settings, disable it immediately so the tile or area is unavailable and the user can't select it. Try not to overuse the disabling tiles feature. Too many unavailable tiles can be distracting.

If a tile displays a value, disabling the tile shouldn't affect the value. The tile should display the same value when it is enabled again. Values that change magically create more work for the user, which is annoying and distracting.

## Nesting Dialog Boxes

AutoCAD limits the number of nested dialog boxes to eight. For information on how to handle nested dialog boxes, see “Nesting Dialog Boxes” on page 376.

Insert an ellipsis (..) in the label of a button that displays a nested dialog box unless the nested dialog box is an alert box. Don't nest dialog boxes more than three deep. Four levels of nesting are reasonable if the fourth-level dialog boxes are only alert boxes. Because dialog boxes appear initially in the center of the screen, make the nested dialog boxes smaller than the main dialog box.

## Hiding Dialog Boxes

If a user needs to make a selection from the graphics screen before the dialog box has closed, you must close the dialog box momentarily so that the user can see the screen and make the selection. Once the selection is made, you need to display the dialog box again. This is known as hiding a dialog box.

The label of a button that causes the dialog box to be hidden should not contain an ellipsis. Instead, use a space followed by a less-than symbol (<) in the label. When the dialog box hides itself, a prompt should be displayed that explains what the user is expected to do.

In most cases you can get the input with one of the `getxxx` functions. These functions have an argument with which you can specify a prompt.

When the dialog box reappears, it should contain feedback on the selection process. This can be new information in the edit box fields, an updated list box, a text message that indicates the status, or a combination of these.

## Providing Defaults

Provide reasonable defaults for all entries and options. Well-chosen defaults can help users complete a dialog box quickly and easily.

It is recommended that you update the default values—in other words, that you save the user's previous settings and use them as the new defaults—each time the dialog box is used. Even if the user has to change some of these, it is less work than starting from the beginning each time.

## Handling Keyboard Input

When you create a custom dialog box, you can specify how it handles keyboard shortcuts. Some keyboard shortcuts are common to all dialog boxes. For example, the TAB key generally enables users to move from tile to tile, and the SPACEBAR allows users to turn toggles off or on. Each active tile should be a tab stop (the default).

Two keys commonly act as accelerator keys. The accept key (usually the ENTER key) accepts the dialog box and values entered. The cancel key (ESC), discards the dialog box and the values. When AutoCAD first displays a dialog box, one of its tiles has the initial keyboard focus. What the user enters affects this tile until the user moves focus to another tile.

To move from tile to tile, the user can press TAB, choose another tile, or enter one of the keyboard shortcuts known as mnemonic keystrokes. For example, in the Drawing Aids dialog box you can select one of the Isometric Snap/Grid radio buttons by pressing L, T, or R. Moving between tiles changes focus but doesn't make a selection. To select a tile that has focus, users must press the Accept key. For some kinds of tiles, a double-click is equivalent to pressing the Accept key.

## International Language Considerations

Words in other languages often require more characters than their English equivalents. If your dialog boxes are going to be translated, leave as much room as possible for them to grow. The following table shows some familiar AutoCAD terms with their equivalents in French and German.

## Equivalent AutoCAD terms in English, French, and German

English	French	German
Line	Ligne	Linie
Arc	Arc	Bogen
Circle	Cercle	Kreis
3D Polylines	Polylignes 3D	3D-Polylinien
Diameter dimensioning	Cotation de diamètre	Durchmesserbemaßung
Layers	Calque	Layer
Linetypes	Types de ligne	Linientypen
Entity creation modes	Modes de création des objets	Modus für Objekterzeugung
Select objects	Choix des objets	Objekte wählen
OK	OK	OK
Cancel	Annuler	Abbruch
Help	Aide	Hilfe

## Guidelines for Predefined Tiles and Clusters

This section lists recommended conventions and design guidelines associated with particular kinds of predefined tiles and tile clusters.

### Buttons

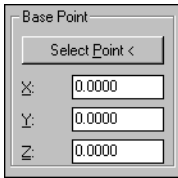
The action associated with a button should be visible to the user and should take place immediately. The label of a button should be unambiguous. Usually, it should be a verb that describes the effect of pushing the button, though another label—such as OK or Options—is acceptable if its meaning is clear. For buttons that call other dialog boxes or hide the current dialog box, see “Nesting Dialog Boxes” on page 358 and “Hiding Dialog Boxes” on page 358.

Buttons in a column should be the same width. In other cases, buttons should have a fixed width (either `fixed_width = true;` , or `children_fixed_width = true;` ) in their common parent cluster.

## Clusters

A boxed cluster (a row or column) is called a group box or an area. An area provides a visual cue to users by isolating and naming controls that work together. The area can contain as many tiles, rows, and columns (unboxed) as necessary. The label of an area should indicate its purpose.

If controls relate to each other, put them in an area. The Base Point cluster in the Block Definition sample dialog box demonstrates this technique with an area formed from a cluster with a label and a border:



However, do not overuse areas. White space is also an effective way to group tiles. Do not put a box around a list box; this looks too busy.

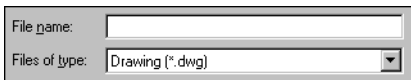
## Edit Boxes

The length of the text-entry portion of an edit box should roughly equal the length of an average entry. When in doubt, use a character width of 10 for real number fields and 20 for text fields.

The label of an edit box should end with a colon (:).

If there are restrictions on what users can enter in the edit box, put a text tile to the right of the edit box that briefly explains these restrictions. If users need to enter a file name, for example, there's no need to explain what a file name is. But if the string is a number that cannot exceed 100, a reminder of this limit is a good idea.

For data, such as points, provide two or three edit boxes rather than require users to remember the Command line syntax of point entry. One exception is an edit box intended specifically for entering advanced syntax, such as the wild-card pattern edit box in the AutoCAD File Search dialog box.



## Image Buttons and Image Tiles

If you use an image button or image tile as an icon to alert the user—for example, a warning signal such as a stop sign—use it consistently in all dialog boxes.

When you use image buttons to represent selections, supplement the image with text that briefly describes it, especially if the color of the image (or part of the image) is a factor in selecting it.

## List Boxes

Because DCL list boxes cannot be scrolled horizontally, the width of the list box should accommodate the longest item in the list. Provide a label (or a text tile) to explain the contents of the list box, unless the list box is the main tile in the dialog box. In that case the dialog box's label might be sufficient—although you must give the list box a label if you want users to be able to move to that list box by using a mnemonic.

Alphabetize the items in the list unless you have a logical reason to organize them in some other way. If the length of the list is fixed and short, consider using a radio column instead of a list box.

## Radio Buttons, Radio Rows, and Radio Columns

Radio columns look better and are easier to use than radio rows. Radio rows are appropriate only when they contain a small number of buttons (usually, two to four) or if the labels are short.

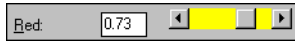


If an option selected elsewhere makes the choices in the radio row or radio column invalid or irrelevant, then disable the whole row or column. In some situations, an option selected elsewhere may make certain radio buttons invalid or irrelevant. In situations like this, you can disable buttons individually.

## Sliders

The granularity of a slider should not be too coarse. For example, if a slider is assigned only four incremental values but is laid out in a two-inch section of the dialog box, users would have to move half an inch to see a change. Avoid jumpiness like this by scaling the size of the slider.

If users need to know the value controlled by the slider, your dialog box should also display the slider's current value. Update this value whenever the slider is moved. It is recommended you also display an edit box that enables users to enter the value rather than use the slider. If you use an edit box this way, update its value; otherwise, display the value in a text tile. The following figure shows a typical combination of slider and edit box:



## Text

When labels are not sufficient, use text tiles to identify the purpose of individual tiles or dialog box areas. You can also use text tiles to display status messages or reminders, including error messages and warnings.

Text should be direct and unambiguous. Describe options and entry fields in terms your users would use. For example, the error message “Invalid entry” in a list box conveys little information. A message such as “Layer does not exist” is more helpful.

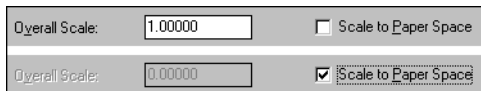
Align messages with the control tiles they describe.

Put text that identifies a group of control tiles or a section of the dialog box above the tiles that the text describes.

## Toggles

When the options controlled by toggles relate to the same topic, group them together.

Use a single toggle that controls whether other tiles, often in a row or column, are active. In this case, the toggle should be prominent. When the toggle controls only one other tile, you can also place it to the right of that tile. The toggle in the following dialog box enables or disables another tile:



## Error Handling

Dialog boxes can display error messages and warnings with a text tile known as an error tile (errtile), or with a nested alert box. The following guidelines apply to both:

- Error messages should be complete sentences, punctuated as such, with an initial capital and a period at the end.
- Error messages should explain clearly the problem or potential problem.
- After reporting the error, shift the dialog box’s focus to the tile that triggered the error, if possible.

## Error Tiles

Use an error tile for minor errors or warnings, especially those that arise from typos and other input errors.

Do not display errors in text tiles used for status messages. These are easy to overlook.

Error tiles should appear at the bottom of a dialog box. Use the standard error tile described in “Dialog Box Exit Buttons and Error Tiles” on page 410.

## Alert Boxes

You can display a standard alert box with a single OK button by calling the `alert` function. Use alert boxes for serious or potentially fatal errors, but do not overuse them. Alert boxes require user input. Therefore, they can be annoying, especially when they report minor errors or obscure the entry that needs to be corrected.

Use alert boxes to warn users that the action about to begin can destroy data or can be time consuming. Alert boxes of this sort should give users a choice of proceeding or canceling the action. If the alert box offers users a choice, such as Proceed or Cancel, you must construct it yourself.

If the alert box provides users with a choice, the text in the alert box should first describe the problem and then pose the next action as a question. In such cases it is important that the button for proceeding be labeled with a verb that describes what will happen. In this context, Overwrite, for example, is less ambiguous than OK, and is an aid to experienced users who will gloss over the text because they have seen this alert box many times.

Unless the error is truly fatal, provide a way for users to return to a previous step or escape from the operation that triggered the alert box.

Usually the default button for a dialog box is OK or its equivalent, but when the situation described by the alert box has serious consequences, make Cancel or its equivalent the default.



# Managing Dialog Boxes

With AutoCAD, you can design and implement dialog boxes to use with your applications. The appearance of a dialog box is defined by dialog control language (DCL) files, as described in “Designing Dialog Boxes.” You control the functionality of a dialog box with an AutoLISP application. This chapter describes how to control dialog boxes using AutoLISP. Although this chapter shows some examples of DCL files, you may find it helpful to read “Designing Dialog Boxes” before reading this chapter.



## In This Chapter

# 12

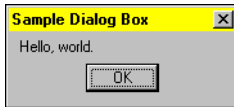
- Controlling Dialog Boxes with AutoLISP Programs
- Action Expressions and Callbacks
- Handling Tiles
- Nesting Dialog Boxes
- Hiding Dialog Boxes
- List Boxes and Pop-Up Lists
- Image Tiles and Buttons
- Application-Specific Data
- DCL Error Handling
- Dialog Box Function Summary

# Controlling Dialog Boxes with AutoLISP Programs

This chapter begins with an overview of the process you use to display dialog boxes and respond to user input from an AutoLISP program.

## Quick Overview

This example starts with a simple dialog box:



The following DCL defines the dialog box:

```
hello : dialog {  
  label = "Sample Dialog Box";  
  : text { label = "Hello, world."; }  
  ok_only;  
}
```

This DCL defines a dialog box labeled **Sample Dialog Box** that contains a text tile and a single OK button. The DCL resides in a file named *hello.dcl*.

**To display the dialog box and respond to the user pressing OK**

- 1 Use the `load_dialog` function to load the DCL file into memory. For example:

```
(setq dcl_id (load_dialog "hello.dcl"))
```

The `load_dialog` function returns a DCL identification number. You need this to identify the dialog in subsequent function calls.

- 2 Call the `new_dialog` function and pass it the dialog name and DCL identification number as arguments, as follows:

```
(new_dialog "hello" dcl_id)
```

- 3 Initialize the dialog box by setting up tile values, lists, and images.

The DCL example above uses a predefined tile named `ok_only`, so you do not have to initialize the tile unless you want to override its default values. The `ok_only` tile also has an action named `done_dialog` assigned to it. If the user presses the OK button, AutoCAD passes the `done_dialog` call to your AutoLISP application and ends the dialog.

- 4 Call `start_dialog` to pass control of the dialog to AutoCAD for display to the user:

```
(start_dialog)
```

- 5 Call `unload_dialog` to remove the dialog from memory after the user responds.

Steps 3, 4, and 5 are dependent on the `new_dialog` function returning a non-`nil` value. For the sake of simplicity, no error processing is included in this example.

You can use the following function to call the sample dialog box:

```
(defun C:HELLO( / dcl_id )
  (setq dcl_id (load_dialog "hello.dcl")) ; Load the DCL file.
  (if (not (new_dialog "hello" dcl_id)) ; Initialize the dialog.
      (exit) ; Exit if this doesn't
              ; work.
  )
  (start_dialog) ; Display the dialog
                  ; box.
  (unload_dialog dcl_id) ; Unload the DCL file.
  (princ)
)
```

Enter this code into a new VLISP text editor window and load the program by choosing Tools ► Load Text in Editor from the VLISP menu. To display the dialog box, enter `(c:hello)` at the VLISP Console prompt.

Note that the `start_dialog` call remains active until the user selects a tile (usually a button) whose associated action expression calls `done_dialog`. The `done_dialog` call can be issued explicitly by the tile. The `done_dialog` call is also issued by the selected tile if its `is_cancel` attribute is set to `true`.

---

**WARNING!** In theory, the dialog box facility takes control of input at the time you call `start_dialog`, but in Windows it takes control when you call `new_dialog`. This has no effect on writing programs. However, if you invoke these functions interactively (at the AutoCAD Command prompt or a VLISP window), you must enter them as one statement. Enclose them within a `progn` or another function. If you don't, the interactive call to `new_dialog` can freeze the screen. Calling `new_dialog` and `start_dialog` interactively can be useful during debugging. (For an example of using these functions interactively, see "DCL Error Handling" on page 386.)

---

## Functions Restricted When a Dialog Box Is Open

While a dialog box is active—that is, during the `start_dialog` call—you cannot call any function that requires user input on the AutoCAD command line, or affects the display outside the dialog box (for example, in the

AutoCAD graphics window). This restriction includes functions that write text, such as `print`, `princ`, and `prin1`.

You can issue `ssget` calls, as long as you do not use any options that require user input.

If your program calls one of the restricted functions between the `start_dialog` and `done_dialog` calls, AutoCAD terminates all dialog boxes and displays the following error message:

AutoCAD rejected function

You can test the `CMDACTIVE` system variable to determine if a dialog box is active. If `CMDACTIVE` is greater than 7, a dialog box is active. The `CMDACTIVE` system variable has bit-coded values that indicate command, script, and dialog box activity.

---

**NOTE** If your application requires users to enter input based on the graphics screen rather than on the dialog box itself (for example, to specify a point or select an object), you must hide the dialog box. That is, you must call `done_dialog` so the graphics screen is visible again, and then restart the dialog box after the user has made the selection. For more information, see “Hiding Dialog Boxes” on page 377.

---

The `term_dialog` function terminates all current dialog boxes as if the user had canceled each of them. This function can be used to cancel a series of nested dialog boxes.

## Action Expressions and Callbacks

To define the action taken when a certain tile in a dialog box is selected, you associate an AutoLISP expression with that tile by calling the `action_tile` function. This is known as an action expression. Within the action expression, you often need access to attributes in the DCL file. The `get_tile` and `get_attr` functions provide this capability. The `get_attr` function retrieves the user-defined attributes within the DCL file. The `get_tile` function gets the current runtime value of a tile based on user input to that tile. Action expressions must be defined following the `new_dialog` call and before the `start_dialog` call.

Information relating to how the user has selected a tile or modified a tile's contents is returned to the action expression as a callback. In most cases, every active tile within a dialog box can generate a callback. As with reactors, the action expression that responds to the callback is often referred to as a

callback function. This function should perform validity checking on the associated tile and should update information in the dialog box that pertains to the value of the tile. Updating the dialog box can include issuing an error message, disabling other tiles, and displaying the appropriate text in an edit box or list box.

Only the OK button (or its equivalent) should query the tile values to permanently save the settings that the user has finally selected. In other words, you should update the variables associated with tile values within the callback for the OK button, not the callback for an individual tile. If permanent variables are updated within the individual tile callbacks, there is no way to reset the values if the user selects the Cancel button. If the OK button's callback detects an error, it should display an error message and return focus to the tile in error; it should not exit the dialog box.

When a dialog box includes several tiles whose handling is similar, it can be convenient to associate those tiles with a single callback function. The principle of not committing to the user's changes until the user chooses OK still applies.

There are two ways to define actions other than calling `acti on_t i l e`. You can define a default action for the entire dialog box when you call `new_d i a l o g`, and you can define an action by using a tile's `acti on` attribute. These alternative means of defining actions, and the order in which they occur, are described in "Default and DCL Actions" on page 372.

## Action Expressions

An action expression can access the variables shown in the following table, indicate which tile was selected, and describe the tile's state at the time of the action. The variable names are reserved. Their values are read-only and have no meaning, unless they are accessed within an action expression.

Action expression variables	
Variable	Description
\$key	The key attribute of the tile that was selected. This variable applies to all actions.
\$value	The string form of the current value of the tile, such as the string from an edit box, or a "1" or "0" from a toggle. This variable applies to all actions.  If the tile is a list box (or pop-up list) and no item is selected, the \$value variable will be nil.

## Action expression variables (*continued*)

Variable	Description
<code>\$data</code>	The application-managed data (if any) that was set just after <code>new_dialog</code> time by means of <code>client_data_tile</code> . This variable applies to all actions, but <code>\$data</code> has no meaning unless your application has already initialized it by calling <code>client_data_tile</code> . See “Application-Specific Data” on page 386.
<code>\$reason</code>	The reason code that indicates which user action triggered the action. Used with <code>edit_box</code> , <code>list_box</code> , <code>image_button</code> , and <code>slider</code> tiles. This variable indicates why the action occurred. Its value is set for any kind of action, but you need to inspect it only when the action is associated with an <code>edit_box</code> , <code>list_box</code> , <code>image_button</code> , or <code>slider</code> tile. See “Callback Reasons” in the following section for details.

If `edit1` is a text box, the action expression in the following `action_tile` call is evaluated when the user exits the text box:

```
(action_tile "edit1" "(setq ns $value)")
```

The `$value` contains the string that the user entered, and the expression saves this in the `ns` variable.

The next example saves the name of the selected tile so that the program can refer to it:

```
(action_tile "edit1" "(setq newtile $key)")
```

The `newtile` variable is set to the key name of the selected tile, in this case “`edit1`”. The `$key` variable is very useful within a function that serves as the action for several separate tiles.

When a tile is named in more than one `action_tile` call, only the last such call (prior to `start_dialog`) has any effect. (It’s as if you were to assign multiple values to the same variable.) The programmable dialog box (PDB) feature allows only one action per tile.

## Callback Reasons

The callback reason, returned in the `$reason` variable, specifies why the action occurred. Its value is set for any kind of action, but you need to inspect

it only when the action is associated with an `edit_box`, `list_box`, `image_button`, or `slider` tile. The following table shows the possible values:

Callback reason codes	
Code	Description
1	This is the value for most action tiles. The user has selected the tile (possibly by pressing <b>ENTER</b> , if the tile is the default and the platform recognizes accelerator keys).
2	Edit boxes: The user has exited the edit box, but has not made a final selection.
3	Sliders: The user has changed the value of the slider by dragging the indicator but has not made a final selection.
4	List boxes and image buttons: This callback reason always follows a code 1. It usually means “commit to the previous selection.” It should never undo the previous selection; this confuses and annoys the user.

Code 1 is described fully in the table. The following text describes the codes 2, 3, and 4 in greater detail.

#### Code 2—Edit Boxes

The user has exited the edit box—by pressing the **TAB** key or by choosing a different tile—but has not made a final selection. If this is the reason for an edit box callback, your application should not update the value of the associated variable, but should check the validity of the value in the edit box.

#### Code 3—Sliders

The user has changed the value of the slider by dragging the indicator (or an equivalent action), but has not made a final selection. If this is the reason for a slider callback, your application should not update the value of the associated variable but should update the text that displays the slider’s status. For more information, see “Sliders” on page 362. For code examples, see “Handling Sliders” on page 375.

#### Code 4—List Boxes

The user has double-clicked on the list box. You can define the meaning of a double-click in your application. If the main purpose of the dialog box is to select a list item, a double-click should make a selection and then exit the dialog box. (In this case, the `is_default` attribute of the `list_box` tile should be true.) If the list box is not the primary tile in the dialog box, then a double-click should be treated as equivalent to making a selection (code 1).

List boxes that allow the user to select multiple items (`multiple_select = true`) cannot support double-clicking.

#### Code 4—Image Buttons

The user has double-clicked on the image button. You can define the meaning of a double-click in your application. In many cases it is appropriate for a single-click to select the button, but in others it is better for a single-click (or a keyboard action) to highlight the button, and then have the ENTER key or a double-click select it.

### Default and DCL Actions

The `action_tile` function is not the only way to specify an action. A tile's DCL description can include an `action` attribute in AutoLISP, and the `new_dialog` call can specify a default action for the dialog box as a whole. A tile can have only a single action at a time. If the DCL and the application specify more than one action, they supersede each other in the following order of priority (lowest to highest):

- The default action specified by the `new_dialog` call (used only if no action is explicitly assigned to the tile).
- The action specified by the `action` attribute in the DCL file.
- The action assigned by the `action_tile` call (highest priority).

## Handling Tiles

Your program has some control over the tiles that are in the current dialog box at initialization time and action (callback) time. This section introduces the tile-handling functions and shows how to initialize and modify the tiles' modes and values.

### Initializing Modes and Values

Initializing a tile can include the following:

- Making it the initial keyboard focus of the dialog box
- Disabling or enabling it
- Highlighting its contents, if it is an edit box or image

These operations are performed by `mode_tile` calls. You can set the value of a tile by using `set_tile`.



To display a default value—such as a surname—in an edit box and set the dialog box’s initial focus to that box, use the following code:

```
(setq name_str "Kenobi ") ; Default.  
(set_tile "lastname" name_str) ; Initializes field.  
(mode_tile "lastname" 2) ; 2 sets focus to tile.
```

An additional `mode_tile` call can highlight all the contents of an edit box, so the user has the option to type immediately over the default contents, as shown in the following example:

```
(mode_tile "lastname" 3) ; 3 selects box contents.
```

## Changing Modes and Values at Callback Time

At callback time, you can check the value of a tile. If necessary, you can use `set_tile` again to modify this value. During callbacks, you can also use `mode_tile` to change the status of a tile. The following table shows the values of the `mode_tile` *mode* argument:

Mode codes for <code>mode_tile</code>	
Value	Description
0	Enable tile
1	Disable tile
2	Set focus to tile
3	Select edit box contents
4	Flip image highlighting on or off

When you use `mode_tile` to disable a tile that has the current focus, you must call `mode_tile` again to set the focus to a different tile (in most cases, the next tab stop in the dialog box). Otherwise, the focus will remain on a disabled tile, which is illogical and can cause errors.

A good example of a tile disabling itself is a series of dialog box pages that the user steps through by choosing a Next or Previous button. When the user chooses Next on the next-to-last page, the button is disabled. The same thing happens after choosing Previous on the second page. In both cases, the code must disable the button that was chosen, and then set focus to a different tile.

Suppose the tile called `group_on` is a toggle that controls a cluster called `group`. When the toggle is turned off, the tiles in the cluster are inactive and should not be modified. In this case, you might define the following action for the toggle. (Notice the use of the `\` control character, which allows quotation marks within an `action_tile` argument.)

```
(action_tile "group_on" "(mode_tile \"group\" (- 1 (atoi $value)))")
```

The subtraction and `atoi` call in the action expression set the `mode_tile` function's `mode` argument. Because a toggle is 0 when it is turned off and 1 when it is turned on, the subtraction inverts its value and the `mode` controls whether the cluster is enabled.

You can inspect attributes other than a tile's value with the `get_attr` function. For example, you may want to retrieve the label of a button called "pressme":

```
(get_attr "pressme" "label ")
```

The `get_attr` function returns the value of the specified attribute as a string.

---

**NOTE** If you use `get_attr` to retrieve a value attribute, it gets the value attribute saved in the DCL file (the initial value of the tile). The `get_tile` function, however, gets the current runtime value of the tile. The two values are not necessarily the same.

---

## Handling Radio Clusters

Radio buttons appear in radio clusters. The value of each radio button is either "1" for On or "0" for Off. The value of the radio cluster is the key attribute of the currently selected button. The PDB feature manages the values of radio buttons in a cluster and ensures that only one button is turned on at a time. You can assign an action to each radio button, but it is more convenient to assign an action to the radio cluster as a whole and then test the cluster's value to see which radio button was chosen.

Consider the following example: A radio cluster controls the view of a three-dimensional object that is displayed after a user exits a dialog box. This cluster contains four radio buttons:

```
(action_tile "view_sel" "(pick_view $value)")
.
.
.
```

```
(defun pick_view (which)
  (cond
    ((= which "front") (setq show_which 0))
    ((= which "top") (setq show_which 1))
    ((= which "left") (setq show_which 2))
    ((= which "right") (setq show_which 3))
  )
)
```

These examples show each radio button associated with a single variable that takes multiple values. These variables may also cause additional actions, such as disabling selections in your dialog box. If the radio cluster is large, you can store the associated values in a table. If you use a table, structure it so it doesn't depend on the order of the buttons within the cluster. The PDB feature does not impose this restriction, and the order can change if the DCL definition changes.

## Handling Sliders

When you handle actions and callbacks from sliders, your application should check the reason code that it receives along with the callback. This is not required, but it is a good idea because it can reduce processing.

A callback occurs when an increment boundary on a slider is crossed. For example, if the slider is defined with a minimum value of 0, a maximum value of 10, and both small and big increments of 1, a callback is issued 10 times as the user traverses from one end of the slider to the other.

The following function shows the basic scheme of a function to handle a slider. It is called from an action expression associated with the slider tile. The `slider_info` tile used by the function displays the slider's current value in decimal form. Often such a tile is an edit box as well, which gives users the choice of either manipulating the slider or entering its value directly. If a user enters a value in `slider_info`, your edit box callback should update the value of the slider as follows:

```
(action_tile
  "myslider"
  "(slider_action $value $reason)"
)
(action_tile
  "slider_info"
  "(ebox_action $value $reason)"
)
.
.
.
```

```

(defun slider_action(val why)
  (if (or (= why 2) (= why 1)) ; Check reason code.
      (set_tile "slider_info" val) ; Show interim result.
      )
  )

(defun ebox_action(val why)
  (if (or (= why 2) (= why 1)) ; Check reason code.
      (set_tile "myslider" val) ; Show interim result.
      )
  )

```

## Handling Edit Boxes

Actions and callbacks to handle edit boxes are similar to those for sliders. However, because characters in edit boxes are already visible, there is no need for action on interim results. Edit boxes only return a callback code when the focus to that tile is lost. The following code example checks the value but doesn't redisplay it:

```

(action_tile "myeditbox" "(edit_action $value $reason)")
.
.
.
(defun edit_action (val why)
  (if (or (= why 2) (= why 4))
      . ; Do range checking on
      . ; transient value here.
      )
  )

```

## Nesting Dialog Boxes

You create and manage nested dialog boxes by calling `new_dialog` and `start_dialog` from within an action expression or callback function. For example, by including the following statement, a function can display the “Hello, world” box when the user chooses the button called `button_1`:

```

(action_tile "button_1" "(c:hello)")

```

The user must exit the nested dialog box before using the previous dialog box again.

AutoCAD imposes a limit of no more than eight nested dialog boxes, but to avoid confusion it is recommended you nest dialog boxes no deeper than four levels.

---

**WARNING!** If you display nested dialog boxes by multiple `new_dialog` calls, be careful to balance each `new_dialog` call with a corresponding `done_dialog` call (whether called from a callback or not). Otherwise, your application may fail.

---

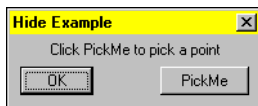
The `term_dialog` function terminates all current dialog boxes as if the user had canceled each of them. You can use this function if you need to cancel a series of nested dialog boxes.

## Hiding Dialog Boxes

A user cannot make an interactive selection while a dialog box is active. If you want the user to make a selection from the graphics screen, you must hide your dialog box and then restore it. Hiding the box is the same as ending it with `done_dialog`, except your callback function must use the `done_dialog` *status* argument to indicate that the dialog box is hidden—as opposed to ended or canceled. Set *status* to an application-defined value.

The `start_dialog` function returns the application-defined *status* when the dialog box disappears. Your program must then examine the status returned by `start_dialog` and determine what to do next. For standard and application-defined *status* values, see “`done_dialog`” in the *AutoLISP Reference*.

For example, here is a simple dialog box that may require a user to pick a point in the AutoCAD graphics window:



The dialog box is defined with the following DCL:

```
hi dedcl : dialog
{ label = "Hi de Exampl e";
  : column
  { : text
    { key = "message";
      label = "Click PickMe to pick a point";
      fixed_width = true;
      fixed_height = true;
      alignment = centered;
    }
  }
}
```

```

: row
{ ok_only;
  : retirement_button
  { label      = "PickMe";
    key        = "hide";
    mnemonic   = "H";
  }}}}

```

The function controlling the dialog box displays the window until the user presses OK or closes the window. If the user chooses **PickMe**, the code hides the dialog box and prompts the user to select a point. The following AutoLISP code controls the dialog box:

```

(defun c:hidedcl (/ dcl_id what_next cnt)
  (setq dcl_id (load_dialog "hidedcl.dcl")) ; Load the dialog box.
  (setq what_next 2)
  (setq cnt 1)
  (while (>= what_next 2) ; Begin display loop.
    (if (null (new_dialog "hidedcl" dcl_id)) ; Initialize dialog
      (exit) ; box, exit if nil
    ); endif ; returned.

    ; Set action to take if a button is pressed. Either button
    ; results in a done_dialog call to close the dialog box.
    ; Each button associates a specific status code with
    ; done_dialog, and this status code is returned by
    ; start_dialog.

    (action_tile "accept" "(done_dialog 1)") ; Set action for OK.
    (action_tile "hide" "(done_dialog 4)") ; Set action for
    ; PickMe.
    (setq what_next (start_dialog)) ; Display dialog box.
    ;
    (cond
      ((= what_next 4) ; Prompt user to
        (getpoint "\npick a point") ; pick pt.
      )
      ((= what_next 0)
        (prompt "\nuser cancelled dialog")
      )
    )
  )
  (unload_dialog dcl_id)
  (princ)
)

```

---

**NOTE** The `term_dialog` function terminates all dialog boxes at once but does not return a status code, so there is no way for an application to distinguish between hiding a nested box and canceling boxes due to an error condition.

---

## Requesting a Password

The following examples show how to use a simple dialog box to request a password from users.

The *getpass.dcl* file defines a dialog box named `passdlg`, which contains two tiles: the `edit_box` tile where the user enters the password, and the `ok_cancel` tile. It uses the `password_char` DCL attribute to mask the text a user enters:

```
// GETPASS. DCL
//
passdlg : dialog {
  label = "Password Protected";
  : edit_box {
    label = "Password: ";
    edit_width = 20;
    key = "password";
    password_char = "?";
  }
  ok_cancel ;
}
```

The *getpass.lsp* file defines the `GETPASS` function. This function loads the *getpass.dcl* file and displays the `passdlg` dialog box. When a user enters text into the edit box, it is masked by the `password_char` character defined in the DCL file. The action assigned to the edit box ensures that the characters entered by the user are set to the `pass` variable:

```
:: GETPASS. LSP
;;
;;
(defun GETPASS ( / dcl_id pass )
  (setq dcl_id (load_dialog "getpass.dcl"))
  (if (new_dialog "passdlg" dcl_id)
    (progn
      (action_tile "password" "(setq pass $value)")
      (start_dialog)
      (unload_dialog dcl_id)
    )
    (princ "Error: Unable to load GETPASS.DCL. ")
  )
  pass
)
```

The `GETPASS` function returns the string entered by the user.

# List Boxes and Pop-Up Lists

The following functions handle list boxes and pop-up lists in dialog boxes:

- `start_list`
- `add_list`
- `end_list`

You set up the lists displayed in list boxes and pop-up lists by using a sequence of calls to these functions.

## List Operations

A dialog box list operation always begins with a `start_list` function call. The function syntax is as follows:

```
(start_list key [operation [index]])
```

The *key* argument is a string that identifies the dialog box tile. The *key* argument is case-sensitive. The *operation* argument is an integer value that indicates whether you are creating a new list, changing a list, or appending to a list. The following are valid *operation* arguments:

### Operation codes for start\_list

Value	Description
1	Change selected list contents
2	Append new list entry
3	Delete old list and create new list (the default)

The *index* argument is only used in change operations. The index indicates the list item to change by a subsequent `add_list` call. The first item in a list is index 0.

If you don't specify *operation*, it defaults to 3 (create a new list). If you do not specify an *index*, the index value defaults to 0.



You implement the list operations as follows:

Creating a New List (3)	After the <code>start_list</code> call, call <code>add_list</code> repeatedly to add new items to the list. End list handling by calling <code>end_list</code> .
----------------------------	--

Changing an Item in a List (1)	After calling <code>start_list</code> , call <code>add_list</code> once to replace the item whose index was specified in the <code>start_list</code> call. (If you call <code>add_list</code> more than once, it replaces the same item again.) End list handling by calling <code>end_list</code> .
--------------------------------------	--

Appending an Item to a List (2)	After calling <code>start_list</code> , call <code>add_list</code> to append an item to the end of the list. If you continue to call <code>add_list</code> , more items are appended until you call <code>end_list</code> .
---------------------------------------	---

Regardless of which list operation you are doing, you must call the three functions in sequence: `start_list`, then `add_list` (possibly more than once), and then `end_list`.

The `mapcar` function is useful for turning a “raw” AutoLISP list into a list box display. In the following example, the `apnames` list contains strings that you want to appear in a list box called `selections`. You can use this code fragment to set up the list and display it as follows:

```
(start_list "selections") ; Specify the name of the list box.  
(mapcar 'add_list apnames) ; Specify the AutoLISP list.  
(end_list)
```

Because list creation (3) is the default, this example doesn't specify it.

The value of a `list_box` tile is the index of the selected item (or the indexes of selected items, if multiple selections are allowed). If your program needs to know the actual text associated with an index, it must save the original list. It must also track changes to the list.

Appending list items is similar to creating a new list. If, for example, `apnames` has 12 items in it, and you want to append another list, called `newnames`, you could use the following code:

```
(start_list "selections" 2)  
(mapcar 'add_list newnames)  
(end_list)
```

Changing a single item requires only one `add_list` call. In this case, you specify the index of the item to change:

```
(start_list "selections" 1 5) ; Change the sixth item in the list.  
(add_list "SURPRISE!") ; Remember that the first index is 0.  
(end_list)
```

You cannot delete a list item or insert an item without rebuilding the list from scratch.

## Processing List Elements

Because the value of a `list_box` tile can contain leading spaces (especially if you are retrieving multiple items), do not test the value as a string comparison. Convert `list_box` value to an integer first with the `atoi` function, before processing the list box. You can also use the `read` function, which converts a token to an integer automatically. For example, for a list named `justone` that accepts only a single selection, the following code fragment checks to see if the third item in the list was selected:

```
(setq index ( get_tile "justone"))
(cond
  ((/= index "") ; See if string is empty.
    (= 2 (atoi index)) ; Process the third entry.
    ...
  )
)
```

It is necessary to first check if the string is empty, because the `atoi` functions return 0 for an empty string as well as the string "0".

The value of a pop-up list never has a leading space, so you don't have to convert the value. Pop-up lists do not allow for multiple selection.

If the list box supports multiple selection, your program must do the conversion and step through the multiple values in the value string. The following definition of `MK_LIST` returns a list containing only items the user has selected from the original `displaylist`. (In this example, the display list `displaylist` is maintained as a global variable.) The `MK_LIST` function expects to be called with the current \$value of the list box:

```
(defun MK_LIST (readlist / count item retlist)
  (setq count 1)
  (while (setq item (read readlist))
    (setq retlist (cons (nth item displaylist) retlist))
    (while (and (/= " " (substr readlist count 1))
      (/= "" (substr readlist count 1)))
      (setq count (1+ count))
    )
    (setq readlist (substr readlist count))
  )
  (reverse retlist)
)
```

Both preceding examples also work for the case of a single selection.

# Image Tiles and Buttons

AutoLISP provides functions for handling image tiles and image buttons. Examples for how to use these functions are provided in this section.

## Creating Images

The calling sequence to create images for image tiles and image buttons is similar to the list-handling sequence. The `start_image` function begins the creation of an image, and `end_image` ends it. However, the type of image to draw is specified in separate function calls, instead of arguments:

<code>vector_image</code>	Draws a vector (a single, straight line) in the current image.
<code>fill_image</code>	Draws a filled rectangle in the current image.
<code>slide_image</code>	Draws an AutoCAD slide in the image.

Vectors and filled rectangles are useful for simple images, such as the color swatches (filled rectangles) that the AutoCAD Select Color dialog box uses to display the user's choice of color. For complicated images, slides are more convenient. However, displaying slides can be time-consuming. If you use slides, keep them simple.

---

**NOTE** If you use slides with filled objects (such as wide polylines, solids, and 3D faces) in image tiles, the images will appear as outlines unless you make the slides from an image created with the `SHADE` command.

---

The `vector_image` function requires that you specify absolute coordinates, while `fill_image` and `slide_image` require that you specify a starting coordinate along with a relative width and height. To do this correctly you must know the exact dimensions of the image tile or image button. Because these dimensions are usually assigned when the dialog box is laid out, the PDB feature provides functions that return the width and height of a particular tile. These dimension functions are `dimx_tile` and `dimy_tile`. You should call them before you begin creating an image. The origin of a tile, (0,0), is always the upper-left corner.

Colors can be specified as AutoCAD color numbers or as one of the logical color numbers shown in the following table. (The values and mnemonics are defined by the Autodesk Device Interface [ADI].)

Dialog box color attribute		
Color number	ADI mnemonic	Meaning
-2	BGLCOLOR	Current background of the AutoCAD graphics screen
-15	DBGLCOLOR	Current dialog box background color
-16	DFGLCOLOR	Current dialog box foreground color (for text)
-18	LINELCOLOR	Current dialog box line color

In the following example, "cur\_color" is an image tile you want to fill entirely with a patch of red as follows:

```
(setq width (dimx_tile "cur_color")
      height (dimy_tile "cur_color"))
(start_image "cur_color")
(fill_image 0 0 width height 1) ; 1 = AutoCAD red.
(end_image)
```

You can use the image-drawing functions in conjunction with each other. The following code fills an image and then draws a vertical stripe over it:

```
(setq width (dimx_tile "stripe")
      height (dimy_tile "stripe"))
(start_image "stripe")
(fill_image 0 0 width height 3) ; 3 = AutoCAD green.
(setq x (/ width 2)) ; Center the vector vertically.
(vector_image x 0 x height 4) ; 4 = AutoCAD cyan.
(end_image)
```

The slides you display with `slide_image` can be standalone slide (SLD) files, or part of a slide library (SLB) file. If the slide is in an SLD file, you specify its name without the *.slid* extension (for example, "frntview"). If the slide is in a slide library, you specify the name of the library, followed by the name of the slide enclosed in parentheses. Note that the library and slide names are also specified without extensions—for example, "allviews(frntview)". The `slide_image` function searches for the slide or slide library file according to the current AutoCAD library search path. (See "load" in the *AutoLISP Reference*.)

In the following example, the slide is in a single file called *topview.sld*:

```
(setq x (dimx_tile "view")
      y (dimy_tile "view"))
(start_image "view")
(slide_image 0 0 x y "topview")
(end_image)
```

Vectors in slides are often drawn in white (color number 7), which is the default background color of an image. If your image tile is blank when you first display a slide, try changing its color attribute to `graphics_background`. (You can also change the background of the image by preceding the `slide_image` call with a `fill_image` call.)

## Handling Image Buttons

You can handle an image button simply as a button—that is, you can use it to trigger a single action. However, you can also use the PDB feature to define regions of the button. With regions defined, the action taken depends on the part of the image button the user selects. The mechanism for this is straightforward: an image button's action or callback returns the (X,Y) location that the user selected. The coordinates are within the range of the particular image button tile (as returned by the dimension functions). Your application must assign a meaning to select locations by implicitly defining regions of the image. The DDVPOINT dialog box makes good use of this feature. You can view this by loading and running the *ddvpoint.lsp* file in the AutoCAD *Support* directory.

In the following example, your image button has two color swatches created by `fill_image`. You want to select either one or the other, depending on which region the user selects. If the image button is divided horizontally (dark above and light below), your action needs to test only the one dimension:

```
(action_tile "image_sel" "(pick_shade $key $value $y)")
...
(defun pick_shade (key val y)
  (setq threshold
    (/ (dimy_tile key) 2)) ;Image is divided horizontally.
  (if (> y threshold)      ;Remember that the origin is at
    (setq result "Light")  ;upper left.
    (setq result "Dark"))
)
```

# Application-Specific Data

The `client_data_tile` function assigns application-specific data to a tile. The data is available at callback time as the `$data` variable and must be a string. Client data is not represented in DCL; it is valid only while your application is running. Using client data is comparable to using user-defined attributes. The main difference is that user-defined attributes are read-only, while client data can change at runtime. Also, end-users can inspect user-defined attributes in the application's DCL file, but client data is invisible to them.

Because your program must maintain the list displayed by a list box (or pop-up list), client data is good for handling this information. The following modification to the `MK_LIST` function (shown in "Processing List Elements" on page 382) makes the list an argument:

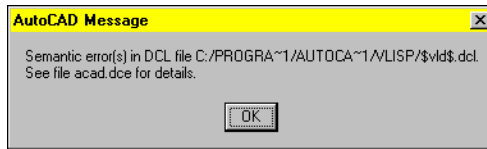
```
(defun MK_LIST (readlist displaylist /)
```

This code eliminates the need for a global list variable. The following calls in the main part of the dialog box handler associate a short list with the tile by calling `client_data_tile`, and then pass that list to `MK_LIST` by means of an action expression as follows:

```
(client_data_tile  
  "colorsyslist"  
  "Red-Green-Blue Cyan-Magenta-Yellow Hue-Saturation-Value"  
)  
(action_tile  
  "colorsyslist"  
  "(setq usrchoice (mk_list $value $data))")  
)
```

## DCL Error Handling

The PDB feature checks a DCL file for errors the first time you load it. If AutoCAD encounters a syntax error, a misuse of attributes, or any other error (such as failure to specify a key attribute for an active tile), the PDB does not load the DCL file. Instead, AutoCAD either displays one or more dialog boxes alerting you to the error, or writes a list of errors to a text file called *acad.dce*. If AutoCAD writes the error messages to *acad.dce*, it alerts you to this with a message similar to the following:



You can inspect the contents of *acad.dce* to find the problem. AutoCAD places the *acad.dce* file in the current working directory. When AutoCAD reads a DCL file successfully, it deletes the *acad.dce* file.

If your application uses multiple DCL files, the *acad.dce* file is overwritten (or deleted if no errors occur) when each new file is loaded. When you test the program, *acad.dce* shows errors (if any) from only the DCL file most recently read. It is recommended that you use the VLISP DCL Preview feature to debug your DCL files (see “Displaying Dialog Boxes with Visual LISP” on page 344). You can also load and debug each file manually in AutoCAD with the `load_dialog` function. The following `load_dialog` function loads the DCL file *hellofile.dcl*:

```
Command: (load_dialog "hellofile")
3
```

If the dialog box loads successfully, `load_dialog` returns a positive integer that identifies the DCL file. You pass this value to the `new_dialog` function to initialize individual dialog boxes in the file.

The `new_dialog` function returns `T` if it succeeds; otherwise it returns `nil`. If `new_dialog` returns `T`, call the `start_dialog` function to display the dialog box.

Once you’ve debugged each DCL file, you can load your program and test the dialog boxes in combination. If your program calls a restricted function between the `start_dialog` and `done_dialog` calls, AutoCAD terminates all dialog boxes and displays the following error message:

AutoCAD rejected function

See “Functions Restricted When a Dialog Box Is Open” on page 367 for information on which functions are restricted.

## Setting the Auditing Level to Affect Error Messages

The level of semantic auditing affects which messages AutoCAD issues for a DCL file (see “Semantic Auditing of DCL Files” on page 346). For example, the `hidedcl` dialog box defined in “Hiding Dialog Boxes” on page 377 dis-

plays without any warnings, if you use the default audit level. If you set the audit level to 3, though, AutoCAD displays an alert dialog box with a warning message. You can see this for yourself by inserting the following line at the beginning of `hi dedcl`:

```
dcl_settings : default t_dcl_settings { audit_level = 3; }
```

Try using the VLISP DCL Preview feature to view the dialog box defined in `hi dedcl`. You will be alerted to view the *acad.dce* file, which contains the following messages:

```
=== DCL semantic audit of C:/PROGRA~1/AUTOCAD~1/VLISP/$vl d$. dcl ===  
Hint in "hi dedcl". (widget type = text, key = "message")  
    fixed_height = true is probably redundant.
```

At lower (less discriminating) levels of semantic auditing, AutoCAD does not look for redundant attribute definitions and the dialog box displays normally.

Remove the `fixed_height = true` statement from the DCL to correct the situation AutoCAD is warning you about.

## Dialog Box Function Summary

This section summarizes the steps required in a typical dialog box handling function. It also describes a sample application you can refer to when designing and implementing your own dialog box functions.

### Function Sequence

The following demonstrates the typical function sequence:

- 1 Load the DCL file with a `load_dialog` call.
- 2 Call `new_dialog` to display a particular dialog box.  
Be sure to check the value returned by `new_dialog`. Calling `start_dialog` when the `new_dialog` call has failed can have unpredictable results.
- 3 Initialize the dialog box by setting up tile values, lists, and images. Initialize also when you call `action_tile` to set up action expressions or callback functions. Other functions typically called at this time are `set_tile` and `mode_tile` for general tile values and states, `start_list`, `add_list`, and `end_list` for list boxes, and the dimension functions with `start_image`, `vector_image`, `fill_image`, `slide_image`, and `end_image` for images. At this time you can also call `client_data_tile` to associate application-specific data with the dialog box and its components.



- 4 Call `start_dialog` to turn control over to the dialog box, so that the user can enter input.
- 5 Process user input from within your actions (callbacks). Process input when you are most likely to use `get_tile`, `get_attr`, `set_tile`, and `mode_tile`.

The user presses an exit button, causing an action to call `done_dialog`, which causes `start_dialog` to return a value. At this point, unload the DCL file by calling `unload_dialog`.

This scheme handles only one dialog box and one DCL file at a time. Applications usually have multiple dialog boxes. The easiest and quickest way to handle these dialog boxes is to save all of them in a single DCL file. The `load_dialog` call then loads all dialog boxes at once, and you can call `new_dialog` for any dialog box. If memory is limited, however, you may need to create multiple DCL files and use `unload_dialog` to remove one set of dialog boxes from memory before you load another set.

## The Sample Block Definition Dialog Box

The sample application *bmake.lsp* and its associated *bmake.dcl* file illustrate a number of useful dialog box techniques. These files are in the AutoCAD *Support* directory. The *bmake* application is essentially an interactive interface to the `entmake` function. You can use it to define new blocks and to view the names of existing blocks. Some of the techniques illustrated by *bmake* are

- Hiding dialog boxes by defining special status codes for `done_dialog` to pass to `start_dialog`. See the main loop of the `C: BMAKE` function (following the `load_dialog` and `action_tile` calls).
- Using a toggle to enable or disable another tile. See the definition of the `DO_UNNAMED` function.
- Building a list for a list box. See the `PAT_MATCH` and `SORT` functions.
- Displaying the standard AutoCAD Help dialog box. See the `DO_HELP` function.

Aside from demonstrating dialog box techniques, *bmake* illustrates good design.



# Programmable Dialog Box Reference

This chapter lists and describes all the dialog control language (DCL) tiles and their associated attributes, and summarizes the VLISP functions available for working with programmable dialog boxes.

## In This Chapter

# 13

- Tile Attributes
- Synopsis of Predefined Attributes
- DCL Attribute Catalog
- Functional Synopsis of DCL Tiles
- DCL Tile Catalog
- Programmable Dialog Box Function Synopsis

# Tile Attributes

A tile’s attributes define its layout and functionality. An attribute is similar to a programming language variable: it consists of a name and a value.

## Attribute Types

The value of an attribute must be one of the following data types:

Integer	Numeric values (both integers and real numbers) that represent distances, such as the width or height of a tile, are expressed in character-width or character-height units.
Real Number	A fractional real number must have a leading digit: for example, 0. 1, not . 1.
Quoted String	A quoted string consists of text enclosed in quotation marks (""). Attribute values are case-sensitive: B1 is not the same as b1. If the string must contain a quotation mark, precede the quotation mark character with a backslash (\"). Quoted strings can contain other control characters as well. The characters recognized by DCL are shown in the following table:

Control characters allowed in DCL strings

Control character	Meaning
\"	quote (embedded)
\\	backslash
\n	newline
\t	horizontal tab

Reserved Word	A reserved word is an identifier made up of alphanumeric characters, beginning with a letter. For example, many attributes require a value of either true or false. Reserved words are also case-sensitive: True does not equal true.
---------------	---

Like reserved words and strings, attribute names are case-sensitive; for example, you cannot assign a width by calling it `Width`.

Application programs always retrieve attributes as strings. If your application uses numeric values, it must convert them to and from string values. For more information on handling tile values within an AutoLISP program, see “Handling Tiles” on page 372.

Some attributes, such as `width` and `height`, are common to all tiles. Attribute specifications are optional. Many attributes have default values that are used if the attribute is not specified. Other attributes are specifically meant for certain kinds of tiles—for example, the background color of an image. If you attempt to assign this attribute to a different kind of tile, AutoCAD may report an error. Usually, it ignores the attribute.

## Restricted Attributes

Do not use the following tile attributes in your DCL files:

- `horizontal_margin`
- `vertical_margin`
- `type`

## User-Defined Attributes

When defining tiles, you can assign your own attributes. The name of the attribute can be any valid name that does not conflict with the standard, predefined attributes described in the previous subsection and summarized in “Synopsis of Predefined Attributes” on page 394. An attribute name, like a keyword, can contain letters, numbers, or the underscore (`_`), and must begin with a letter.

If a user-defined attribute name conflicts with a predefined attribute, the PDB feature does not recognize the attribute as a new one, and attempts to use the value you assign it with the standard attribute. This can be very hard to debug.

The values you assign to the attribute, and their meanings, are defined by your application. Values for user-defined attributes must conform to the types described in “Tile Attributes” on page 392.

Defining attributes is comparable to defining application-specific client data. Both techniques enable the PDB feature to manage data you supply. User-defined attributes are read-only, that is, they are static while the dialog box is active. If you need to change the values dynamically, you must use client

data at runtime. Also, end users can inspect the value of user-defined attributes in the application's DCL file, but client data remains invisible.

The definition of the AutoCAD Drawing Aids dialog box defines its own attribute, `errmsg`, which has a unique string value for each tile. A common error handler uses the value of `errmsg` when it displays a warning. For example, the tile could assign the following value to `errmsg`:

```
errmsg = "Grid Y Spacing";
```

If the user enters an unusable value, such as a negative number, AutoCAD displays the following error message:

Invalid Grid Y Spacing.

The word `Invalid` and the trailing period (`.`) are supplied by the error handler.

User-defined attributes also can be used for limits on the value of a tile and the name of a subdialog box that the tile activates (see “Nesting Dialog Boxes” on page 376).

## Synopsis of Predefined Attributes

This section lists the attributes defined by the PDB feature. The following table summarizes the predefined attributes in alphabetical order. The attributes are described in detail in “User-Defined Attributes” on page 393.

Predefined attributes		
Attribute name	Associated with	Meaning (if specified or true)
<code>action</code>	All active tiles	AutoLISP action expression
<code>alignment</code>	All tiles	Horizontal or vertical position in a cluster
<code>allow_accept</code>	<code>edit_box</code> , <code>image_button</code> , <code>list_box</code>	Activates <code>is_default</code> button when this tile is selected
<code>aspect_ratio</code>	<code>image</code> , <code>image_button</code>	Aspect ratio of an image
<code>big_increment</code>	<code>slider</code>	Incremental distance to move

### Predefined attributes (*continued*)

Attribute name	Associated with	Meaning (if specified or true)
children_alignment	row, column, radio_row, radio_column, boxed_row, boxed_column, boxed_radio_row, boxed_radio_column	Alignment of a cluster's children
children_fixed_height	row, column, radio_row, radio_column, boxed_row, boxed_column, boxed_radio_row, boxed_radio_column	Height of a cluster's children doesn't grow during layout
children_fixed_width	row, column, radio_row, radio_column, boxed_row, boxed_column, boxed_radio_row, boxed_radio_column	Width of a cluster's children doesn't grow during layout
color	image, image_button	Background (fill) color of an image
edit_limit	edit_box	Maximum number of characters users can enter
edit_width	edit_box, popup_list	Width of the edit (input) portion of the tile
fixed_height	All tiles	Height doesn't grow during layout
fixed_width	All tiles	Width doesn't grow during layout
fixed_width_font	list_box, popup_list	Displays text in a fixed pitch font
height	All tiles	Height of the tile
initial_focus	Dialog	Key of the tile with initial focus
is_bold	Text	Displays as bold
is_cancel	Button	Button is activated when the cancel key—usually <b>ESC</b> —is pressed
is_default	Button	Button is activated when the accept key—usually <b>ENTER</b> —is pressed

### Predefined attributes (*continued*)

Attribute name	Associated with	Meaning (if specified or true)
is_enabled	All active tiles	Tile is initially enabled
is_tab_stop	All active tiles	Tile is a tab stop
key	All active tiles	Tile name used by the application
label	boxed_row, boxed_column, boxed_radio_row, boxed_radio_column, button, dialog, edit_box, list_box, popup_list, radio_button, text, toggle	Displayed label of the tile
layout	slider	Whether the slider is horizontal or vertical
list	list_box, popup_list	Initial values to display in list
max_value	slider	Maximum value of a slider
min_value	slider	Minimum value of a slider
mnemonic	all active tiles	Mnemonic character for the tile
multiple_select	list_box	List box allows multiple items to be selected
password_char	edit_box	Masks characters entered in edit_box
small_increment	slider	Incremental distance to move
tabs	list_box, popup_list	Tab stops for list display
tab_truncate	list_box, popup_list	Truncates text that is larger than the associated tab stop
value	Text, active tiles (except buttons and image buttons)	Tile's initial value
width	All tiles	Width of the tile



# DCL Attribute Catalog

DCL tile attributes are described in detail in this section. The attributes are listed alphabetically.

## action

```
action = "(function)";
```

Specifies an AutoLISP expression to perform an action when this tile is selected. Also known as a callback. For some kinds of tiles, an action can also occur when the user switches focus to a different tile.

The possible value is a quoted string that is a valid AutoLISP expression. A tile can have only one action. If the application assigns it an action (with `action_tile`), this overrides the `action` attribute.

---

**NOTE** You cannot call the AutoLISP `command` function from the `action` attribute.

---

## alignment

```
alignment = position;
```

Specifies the horizontal or vertical positioning (justification) of a tile within its cluster.

For a tile that is a child of a column, the possible values are `left`, `right`, or `centered` (default is `left`).

For a tile that is a child of a row, the possible values are `top`, `bottom`, or `centered` (default is `centered`).

You cannot specify the alignment along the long axis of a cluster. The first and last tiles in the cluster always align themselves with the ends of the column or row. Other tiles in the cluster are distributed evenly unless you adjust the distribution by using padding insertion points (see “`spacer_0`” on page 426).

## allow\_accept

`allow_accept = true-false;`

Specifies whether the tile is activated when the user presses the accept key (usually ENTER). If true and the user presses the accept key, the default button (if any) is pressed. The default button is the button tile whose `is_default` attribute is set to true. The `allow_accept` attribute defaults to false.

## aspect\_ratio

`aspect_ratio = real;`

Specifies the ratio of the width of the image to its height (width divided by height). If zero (0.0), the tile is fitted to the size of the image.

Possible values are floating-point values (default: none).

## big\_increment

`big_increment = integer;`

Specifies the value used by the slider's incremental controls. The default value of `big_increment` is one-tenth of the total range. The value must be within the range specified by `min_value` and `max_value`.

## children\_alignment

`children_alignment = position;`

Specifies the default alignment (similar to `alignment`) for all tiles in a cluster. Does not override a child's alignment attribute, if alignment is specified explicitly.

For columns, possible values are left, right, or centered (default: left).

For rows, possible values are top, bottom, or centered (default: centered).

## children\_fixed\_height

`children_fixed_height = true-false;`

Specifies the default height (similar to `height`) for all tiles in a cluster. Does not override a child's height attribute, if it is specified explicitly.

Possible values are `true` or `false` (default: `false`).

---

**NOTE** Use the `fixed_` attributes with discretion. Inconsistent overriding of defaults results in inconsistent layouts.

---

# children\_fixed\_width

`children_fixed_width = true/false;`

Specifies the default width (similar to `width`) for all tiles in a cluster. Does not override a child's width attribute, if it is specified explicitly.

Possible values are `true` or `false` (default: `false`).

---

**NOTE** Use the `fixed_` attributes with discretion. Inconsistent overriding of defaults results in inconsistent layouts.

---

# color

`color = colorname;`

Specifies the background (fill) color of the image. Possible values are an integer or reserved word (default: 7) specified as an AutoCAD color number or as one of the symbolic names shown in the following table:

Symbolic names for colors	
Symbolic name	Meaning
<code>dialog_line</code>	Current dialog box line color
<code>dialog_foreground</code>	Current dialog box foreground color (for text)
<code>dialog_background</code>	Current dialog box background color
<code>graphics_background</code>	Current background of the AutoCAD graphics screen (usually equivalent to 0)
<code>black</code>	AutoCAD color = 0 (black) (appears light on a black background)
<code>red</code>	AutoCAD color = 1 (red)

### Symbolic names for colors (*continued*)

Symbolic name	Meaning
yellow	AutoCAD color = 2 (yellow)
green	AutoCAD color = 3 (green)
cyan	AutoCAD color = 4 (cyan)
blue	AutoCAD color = 5 (blue)
magenta	AutoCAD color = 6 (magenta)
white graphics_foreground	AutoCAD color = 7 (white) (appears black on a light background)

The symbolic names `graphics_background` and `graphics_foreground` are provided as alternatives to the names `black` and `white`. The use of a specific color can be confusing because the color that is actually displayed varies depending on the current AutoCAD configuration. Also, vectors in slides that you display in an image are often drawn in black or white. If your image tile is blank when you first display it, try changing its color to `graphics_background` or `graphics_foreground`.

## edit\_limit

```
edit_limit = integer;
```

Specifies the maximum number of characters a user is allowed to enter in the edit box. A possible value is an integer (default: 132). When the user reaches this limit, AutoCAD rejects additional characters (except for BACKSPACE or DEL). The maximum edit limit allowed is 256 characters.

## edit\_width

```
edit_width = number;
```

Specifies the width in character-width units of the edit (input) portion of the box—the actual boxed portion of the `edit_box` tile. Possible values are an integer or a real number. If `edit_width` is not specified or is zero, and the width of the tile is not fixed, the box expands to fill the available space. If `edit_width` is nonzero, then the box is right-justified within the space occu-

pied by the tile. If it's necessary to stretch the tile for layout purposes, the PDB feature inserts white space between the label and the edit portion of the box.

## fixed\_height

`fixed_height = true-false;`

Specifies if a tile's height is allowed to fill the available space. If this attribute is true, the tile does not fill the extra space that becomes available in the layout/alignment process.

Possible values are true or false (default: false).

## fixed\_width

`fixed_width = true-false;`

Specifies if a tile's width is allowed to fill the available space. If this attribute is true, the tile does not fill the extra space that becomes available in the layout/alignment process.

Possible values are true or false (default: false).

## fixed\_width\_font

`fixed_width_font = true-false;`

Specifies whether a list box or pop-up list will display text in a fixed pitch font. This allows for easier spacing and tab alignment of columns.

Possible values are true or false (default: false).

## height

`height = number;`

Specifies the height of a tile. Possible values are an integer or a real number representing the distance in character height units. Do not specify this value unless the assigned defaults do not have an acceptable appearance. You must specify, however, the height of image tiles and image buttons.

The `height` attribute specifies the minimum height of a tile. This dimension can be expanded when the tile is laid out, unless the height is fixed by one of the `fixed_` attributes. Defaults are dynamically assigned based on layout constraints.

Character-height units are defined as the maximum height of screen characters (including line spacing).

## **initial\_focus**

`initial_focus = "string";`

Specifies the key of the tile within the dialog box that receives the initial keyboard focus. Possible value is a quoted string (no default).

## **is\_bold**

`is_bold = true-false;`

Specifies whether the text is displayed in bold characters. Possible values are `true` or `false` (default: `false`). If `true`, the text is displayed in bold characters.

## **is\_cancel**

`is_cancel = true-false;`

Specifies whether the button is selected when the user presses the ESC key. Possible values are `true` or `false` (default: `false`).

If the action expression for buttons with the `is_cancel` attribute set to `true` does not exit the dialog box (does not call `done_dialog`), the dialog box is automatically terminated after the action expression has been carried out, and the DIASTAT system variable is set to 0.

Only one button in a dialog box can have the `is_cancel` attribute set to `true`.

## **is\_default**

`is_default = true-false;`

Specifies whether the button is the default button selected (“pushed”) when the user presses the accept key. Possible values are `true` or `false` (default: `false`). If the user is in an `edit_box`, `list_box`, or `image_button` that has the

all `low_accept` attribute set to `true`, the default button is also selected if the user presses the accept key or (for list boxes and image buttons) double-clicks. The default button is not selected by the accept key if another button has focus. In this case, the button that has focus is the one selected.

Only one button in a dialog box can have the `is_default` attribute set to `true`.

## is\_enabled

`is_enabled = true-false;`

Specifies whether or not the tile is initially available. Possible values are `true` or `false` (default: `true`). If `false`, the tile is unavailable and appears grayed out.

## is\_tab\_stop

`is_tab_stop = true-false;`

Specifies whether the tile receives keyboard focus when the user moves between tiles by pressing the TAB key. Possible values are `true` or `false` (default: `true`). If the tile is disabled, it is not a tab stop even if this attribute is `true`. If `false`, the tile is not a tab stop.

## key

`key = "string";`

Specifies a name that the program uses to refer to this specific tile. Possible value is a quoted string (no default). Within a particular dialog box, each key value must be unique. This string is case-sensitive: if you specify the key as `BigTile`, you cannot reference it as `bigtile`.

Because the value of a key is not visible to the user, its name can be whatever you choose (as long as it is unique to the dialog box). For the same reason, key attributes do not need to be translated for applications offered in multiple languages.

## label

`label = "string";`

Specifies the text displayed within the tile. Possible value is a quoted string (default: a blank string, " "). The placement of `label` text is tile-specific.

The `label` attribute can specify a mnemonic character for the tile. The mnemonic is underlined in the tile's label.

If a character in the label string is preceded by an ampersand (&), that character becomes the mnemonic. The character doesn't have to be unique to the dialog box. If more than one tile has the same mnemonic, the user presses that key to cycle through the tiles sequentially.

Mnemonics change focus but do not select a tile. If the user specifies a mnemonic key for a tile that contains a group of items, such as a cluster or a list box, the focus is put on the first item in the tile that is a tab stop. Any active tile is a tab stop unless its `is_tab_stop` attribute is set to `false`.

---

**NOTE** The `mnemonic` attribute also specifies a mnemonic character.

---

## layout

`layout = position;`

Specifies the orientation of a slider. Possible values are `horizontal` or `vertical` (default: `horizontal`). For horizontal sliders, the value increases from left to right. For vertical sliders, the value increases from bottom to top.

## list

`list = "string";`

Specifies the initial set of lines (choices) to be placed in the `popup_list` or `list_box`. Possible value is a quoted string (no default). Lines are separated by a new line symbol (`\n`). Tab characters (`\t`) can occur within each line.



## max\_value

`max_value = integer;`

Specifies the upper range of values that a slider returns. Default maximum value is 10000. This value must be a signed, 16-bit integer no greater than 32767.

## min\_value

`min_value = integer;`

Specifies the lower range of values that a slider returns. Default minimum value is 0. This value must be a signed, 16-bit integer no less than -32768. The `min_value` can be greater than the `max_value`.

## mnemonic

`mnemonic = "char";`

Specifies a keyboard mnemonic character for the tile. The mnemonic is underlined in the tile's label. A possible value is a quoted string of a single character (no default). The character must be one of the letters in the tile's label. The character doesn't have to be unique to the dialog box. If more than one tile has the same mnemonic, the user presses that key to cycle through the tiles sequentially.

From the user's point of view, mnemonics aren't case-sensitive. For example, if a button's mnemonic character is *A*, entering *a* or *A* both gives the *A* button focus. However, in the DCL file the mnemonic must be one of the characters in the tile's label, and it must be capitalized as it appears in the `label` string.

Mnemonics change focus. If the user specifies a mnemonic key for a tile that contains a group of items, such as a cluster or a list box, the focus is put on the first item in the tile that is a tab stop. Any active tile is a tab stop unless its `is_tab_stop` attribute is set to `false`.

---

**NOTE** The `label` attribute can also specify a mnemonic character.

---

## multiple\_select

`multiple_select = true-false;`

Specifies whether multiple items in the `list_box` can be selected (highlighted) at the same time. Possible values are `true` or `false` (default: `false`). If `true`, multiple items can be selected at a time.

## password\_char

`password_char = "char";`

Specifies the character to be used to mask user input. If `password_char` is specified and is not null, that character is displayed in the edit box instead of the characters entered by the user. The use of this attribute has no effect on your application's retrieval of the value entered by the user; it alters only the display of the characters in the edit box.

For an example of using the `password_char` attribute in an application, see “Requesting a Password” on page 379.

## small\_increment

`small_increment = integer;`

Specifies the value used by the slider's incremental controls. Default value of `small_increment` is one one-hundredth the total range. The value must be within the range specified by `min_value` and `max_value`. This attribute is optional.

## tabs

`tabs = "string";`

Specifies the placement of tabs in character width units. Possible value is a quoted string containing integers or floating-point numbers, separated by spaces (no default). These values are used for vertically aligning columns of text in a `popup_list` or `list_box`.

For example, the following code specifies a tab stop at every 8 characters.

`tabs = "8 16 24 32";`

## tab\_truncate

`tab_truncate = true-false;`

Specifies whether the text in a list box or pop-up list is truncated if it is larger than the associated tab stop. Possible values are `true` or `false` (default: `false`).

## value

`value = "string";`

Specifies the initial value of a tile. Possible value is a quoted string. The meaning of a tile's value varies depending on the kind of tile. The value of a tile can change at runtime through user input or `set_tile` calls.

The `value` attribute of a tile is not considered when the dialog box is laid out. After the layout is finished and the dialog box has been displayed, `new_dialog` uses the value attributes to initialize each tile in the dialog box. A tile's value attribute has no effect on the size or spacing of tiles in the dialog box.

## width

`width = number;`

Specifies the width of a tile. Possible values are an integer or a real number representing the distance in character-width units. Do not specify this value unless the assigned defaults don't provide acceptable appearance. You must specify, however, the width of image tiles and image buttons.

The `width` of a tile specifies a minimum width. This dimension can be expanded when the tile is laid out unless the width is fixed by one of the `fixed_` attributes. Defaults are dynamically assigned based on layout constraints.

Character width units is defined as the average width of all uppercase and lowercase alphabetic characters, or the screen width divided by 80, whichever is less (average width is  $(\text{width}(\text{A} .. \text{Z}) + \text{width}(\text{a} .. \text{z}))/52$  ).

# Functional Synopsis of DCL Tiles

This section presents the DCL tiles in functional groupings.

## Predefined Active Tiles

The AutoCAD PDB feature has a set of built-in, or predefined, tiles that can be used by themselves or as the basis for more complex tiles. Their definitions appear as comments within the *base.dcl* file. (See “The base.dcl and acad.dcl Files” on page 340.)

When the user chooses an active tile—a button, for example—the dialog box responds by notifying the application controlling the dialog box. Any predefined active tile can have an associated action. The effect of an action can be visible to the user or can be purely internal (for example, a status update). Actions are accompanied by a reason code that indicates what triggered the action. The meaning of the reason depends on which kind of tile triggered it. The following tiles are selectable, active tiles:

<code>button</code>	<code>popup_list</code>
<code>edit_box</code>	<code>radio_button</code>
<code>image_button</code>	<code>slider</code>
<code>list_box</code>	<code>toggle</code>

## Tile Clusters

You can group tiles into composite rows or columns (known collectively as clusters). For layout purposes, a cluster is treated as a single tile. The row or column can be boxed, with an optional label (a cluster without a box cannot be labeled).

Users cannot select a cluster, only individual tiles within the cluster. Clusters cannot have actions assigned to them, with the exception of radio rows and radio columns. The following tiles define clusters:

<code>boxed_column</code>	<code>dialog</code>
<code>boxed_radio_column</code>	<code>radio_column</code>

boxed_radio_row	radio_row
boxed_row	row
column	

## Decorative and Informative Tiles

The tiles listed below do not cause actions and cannot be selected. They are provided to display information or for visual emphasis, or to assist you in laying out the dialog box.

image	spacer_0
text	spacer_1
spacer	

## Text Clusters

A text tile is surrounded by margin space (like any other kind of tile), which presents a problem when you want to combine pieces of text. For example, assume you want to display the following message:

The time is now 0800 hours and 37 seconds.

The actual values (0800 and 37) are supplied by your program. You can do this by creating a concatenated line of text built out of `text_part` tiles. You can also use text parts vertically to create a paragraph that doesn't have too much space between the lines.

The following text cluster tiles are prototypes defined in the *base.dcl* file.

concatenation	text_part
paragraph	

## Dialog Box Exit Buttons and Error Tiles

The *base.dcl* file provides standard button subassemblies for exiting (or “retiring”) a dialog box. Use these standard versions to maintain a consistent appearance across applications.

<code>errtile</code>	<code>ok_cancel_help</code>
<code>ok_only</code>	<code>ok_cancel_help_errtile</code>
<code>ok_cancel</code>	<code>ok_cancel_help_info</code>

You can customize the text in these buttons by using the prototype `retirement_button` as described in “Customizing Exit Button Text” on page 351.

## Restricted Tiles

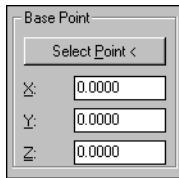
Your DCL files should not use the `tiles_cluster` or `tile`. Also, do not use the basic exit button types (`cancel_button`, `help_button`, `info_button`, and `ok_button`) unless you redefine the standard exit button subassemblies as described in this section in “Dialog Box Exit Buttons and Error Tiles” on page 410.

## DCL Tile Catalog

This section describes all the predefined DCL tiles. The syntax statement, which follows the tile name, lists all the attributes associated with that tile. Any specific attribute functionality is noted following the tile’s description.

### **boxed\_column**

```
: boxed_column {  
    alignment children_alignment  
    children_fixed_height children_fixed_width  
    fixed_height fixed_width height label width  
}
```



A boxed column has a border around it. A dialog box is laid out like a boxed column. If a boxed column is assigned a label, the label appears embedded in the top border. If the label is absent, blank (" "), or null (""), only the box is displayed.

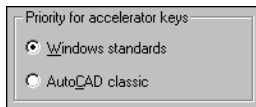
label	Appears as a title. Spacing between a blank and a null label might be different. (See “Fixing the Spacing Around a Boxed Row or Column” on page 350.)
-------	---

## boxed\_radio\_column

```

: boxed_radio_column {
    alignment children_alignment
    children_fixed_height children_fixed_width
    fixed_height fixed_width height label width
}

```



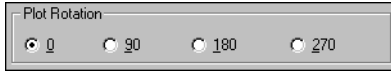
A boxed radio column has a border around it. Treat the label the same way that you would treat the label of a boxed column.

label	Appears as a title. If the label is absent, blank (the default), or null (""), only the box appears. Spacing between a blank and a null label might be different. (See “Fixing the Spacing Around a Boxed Row or Column” on page 350.)
-------	--

value	Specifies the key of the currently selected radio button (the one whose value is "1").
-------	--

## boxed\_radio\_row

```
: boxed_radio_row {  
    alignment children_alignment  
    children_fixed_height children_fixed_width  
    fixed_height fixed_width height label width  
}
```

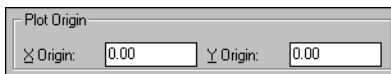


A boxed radio row has a border around it. You treat the label the same way that you would treat the label of a boxed row.

label	Appears as a title. If the label is absent, blank (the default), or null (" "), only the box appears. Spacing between a blank and a null label might be different. (See “Fixing the Spacing Around a Boxed Row or Column” on page 350.)
value	Specifies the key of the currently selected radio button (the one whose value is "1").

## boxed\_row

```
: boxed_row {  
    alignment children_alignment  
    children_fixed_height children_fixed_width  
    fixed_height fixed_width height label width  
}
```



A boxed row has a border around it. If a boxed row has a label, the label appears embedded in it.

label	Appears as a title. If the label is absent, blank (the default), or null (" "), only the box appears. Spacing between a blank and a null label might be different. (See “Fixing the Spacing Around a Boxed Row or Column” on page 350.)
-------	---



## button

```
: button {  
    action alignment fixed_height fixed_width  
    height is_cancel is_default is_enabled  
    is_tab_stop key_label mnemonic width  
}
```



A button tile resembles a push button. The button's `label` specifies text that appears inside the button. Buttons are appropriate for actions that are immediately visible to the user such as leaving the dialog box, or going into a sub-dialog box.

Dialog boxes must include an OK button (or its equivalent) for the user to press after using (or reading) the box. Many dialog boxes also include a Cancel button that enables the user to leave the dialog box without making any changes.

Dialog boxes should use the standard exit button subassemblies described in “Dialog Box Exit Buttons and Error Tiles” on page 410. These subassemblies guarantee that the attributes described in this section are correctly assigned.

---

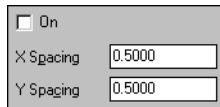
**NOTE** If you make the default button and the cancel button the same, you must make sure at least one other exit button is associated with an action that calls `done_dialog`. Otherwise, the dialog box is always canceled.

---

<code>label</code>	Specifies the text that appears in the button.
--------------------	--

## column

```
: column {  
    alignment children_alignment  
    children_fixed_height children_fixed_width  
    fixed_height fixed_width height_label width  
}
```



Tiles in a column are laid out vertically in the order in which they appear in the DCL file. A column can contain any kind of tile (except for solitary radio buttons), including rows and other columns.

A column without a box has no additional attributes beyond the standard layout attributes.

## concatenation

```
: concatenation {  
    }
```

A concatenation is a line of text made up of multiple, concatenated text\_part tiles. This is useful when you want to insert text that can change at runtime into a standard message. There is a margin around the concatenation as a whole.

The concatenation tile is defined in the *base.dcl* file. See “paragraph” on page 419 for an example that uses concatenation.

## dialog

```
: dialog {  
    initial_focus label value  
}
```

A dialog is the tile that defines the dialog box. You should not specify both a label and value attribute: the value attribute overrides the label attribute.

**label** Specifies the optional title displayed in the title bar of the dialog box.

**value** Specifies a string to display as the optional dialog box title. However, the value isn’t inspected at layout time, so if you assign the title this way, make sure the dialog box is wide enough or the text might be truncated.

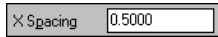
For a dialog, the label and value are equivalent except for layout considerations. To change the title at runtime, use the set\_title function (see “set\_title” in the *AutoLISP Reference*).

**initial\_focus**

Specifies the key of the tile that receives the initial keyboard focus.

## edit\_box

```
: edit_box {  
    action alignment allow_accept edit_limit  
    edit_width fixed_height fixed_width height  
    is_enabled is_tab_stop key_label mnemonic  
    value width password_char  
}
```



An edit box is a field that enables the user to enter or edit a single line of text. An optional `label` can appear to the left of the box. If the entered text is longer than the length of the edit box, the edit box scrolls horizontally.

Left-justifying the `label` and right-justifying the edit box makes it easier to align `edit_box` tiles vertically.

<code>label</code>	Appears as a title. If specified, the label is left-justified within the width of the <code>edit_box</code> tile.
<code>value</code>	The initial ASCII value placed in the box. It is displayed left-justified within the edit (input) part of the box. The value of an edit box is terminated by the null character. If the user enters more characters than the <code>edit_limit</code> and the string is truncated, the null character is appended.

## errtile

```
errtile;
```

An error tile is a text tile that appears at the bottom of a dialog box. By default it is blank, but programs can display messages in it by setting the value of the tile whose key is "error". For example:

```
(set_tile "error" "You can only select one option")
```

The `errtile` tile is defined in the *base.dcl* file.

## image

```
: image {  
    action alignment aspect_ratio color  
    fixed_height fixed_width height is_enabled  
    is_tab_stop key mnemonic value width  
}
```

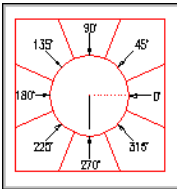


An image is a rectangle in which a vector graphic picture is displayed. Images are used to display icons, linetypes, text fonts, and color patches in AutoCAD dialog boxes. See “Creating Images” on page 383 for instructions on how to generate images for image tiles.

You must assign an image tile either an explicit `width` and `height` attribute, or one of those attributes plus an `aspect_ratio`.

## image\_button

```
: image_button {  
    action alignment allow_accept aspect_ratio  
    color fixed_height fixed_width height  
    is_enabled is_tab_stop key mnemonic width  
}
```



The image button tile is a button that displays a graphic image rather than a label.

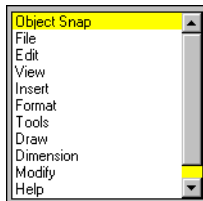
When the user selects an image button, the program obtains the coordinates of the point that was selected. This is useful if you want to display a miniature drawing and assign different meanings to selecting different regions in it.

See “Creating Images” on page 383 for instructions on how to generate images for image buttons.

You must assign an image button either an explicit `width` and `height` attribute, or one of those attributes plus an `aspect_ratio`.

## list\_box

```
: list_box {  
    action alignment allow_accept fixed_height  
    fixed_width height is_enabled is_tab_stop  
    key_label list mnemonic multiple_select tabs  
    value width  
}
```



A list box contains a list of text strings, arranged in rows. Usually the list is of variable length, but list boxes can be used for fixed-length lists when a different kind of tile, such as a set of radio buttons, takes up too much space in the dialog box. When users select a row, it is highlighted. A list box can contain more rows than can fit in the box, so a scroll bar always appears to the right of the list box. (The scroll bar is enabled only if the list has more items than can appear at once.) By dragging the scroll bar cursor or clicking on its arrows, users can scroll through the list box items. Some applications may allow users to select multiple rows.

See “List Boxes and Pop-Up Lists” on page 380 for instructions on how to manage lists for list boxes and pop-up lists.

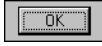
`label`                      Text displayed above the list box.

`value`                    A quoted string containing zero ("" ) or more integers, separated by spaces (no default). Each integer is a zero-based index that indicates a list item that is initially selected. If `multiple_select` is false, `value` cannot contain more than one integer.

If the value string is empty (""), then no items in the list are initially selected. In this case, you don’t need to specify the `value` attribute at all.

## ok\_only

`ok_only;`



The `ok_only` tile is a solitary OK button, such as the kind that alert boxes use. The key of the OK button is "accept".

The `ok_only` tile is defined in the *base.dcl* file.

## ok\_cancel

`ok_cancel ;`

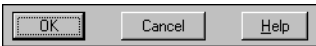


The `ok_cancel` tile is a combination of the OK and Cancel buttons, and is the standard combination for dialog boxes that can originate changes to data. The key of the Cancel button is "cancel".

The `ok_cancel` tile is defined in the *base.dcl* file.

## ok\_cancel\_help

`ok_cancel_help;`



This tile is the `ok_cancel` cluster combined with the Help button. The key of the Help button is "help". Help buttons are recommended for the main dialog box of an application and for complex dialog boxes. The function that handles the Help button can display the standard AutoCAD Help dialog box by invoking the AutoLISP `help` function.

The `ok_cancel_help` tile is defined in the *base.dcl* file.

## ok\_cancel\_help\_errtile

`ok_cancel_help_errtile;`

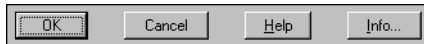


The `ok_cancel_help_errtile` tile provides a convenient way to specify the exit buttons and error tile all at once.

The `ok_cancel_help_errtile` tile is defined in the *base.dcl* file.

## ok\_cancel\_help\_info

`ok_cancel_help_info;`

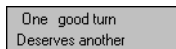


The `ok_cancel_help_info` tile does everything that the `ok_cancel_help` tile does, but it also includes an information button for displaying additional material. It might display the name of your application, the logo of your firm, the application's version number, how to obtain support, and so on. The key of the Info button is "info".

The `ok_cancel_help_info` tile is defined in the *base.dcl* file.

## paragraph

```
: paragraph {
}
```



A paragraph is a cluster of `text_part` or `concatenation` tiles that are arranged vertically. You can construct paragraphs of running text either statically or at runtime. There is a margin around the paragraph as a whole.

The paragraph tile is defined in the *base.dcl* file.

The illustration above was generated with the following DCL:

```
: paragraph
{
  : concatenation
  {
    : text_part
    {
      label = "One";
    }
  }
}
```

```

        : text_part
        {
            label = "good turn";
        }
    }
    : text_part {
        label = "Deserves another";
    }
}

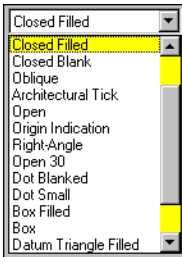
```

## popup\_list

```

: popup_list {
    action alignment edit_width fixed_height
    fixed_width height is_enabled is_tab_stop
    key_label list mnemonic tabs value width
}

```



A pop-up list, or simply pop-up, is functionally equivalent to a list box. When a dialog box is first displayed, the pop-up is in a collapsed state and looks like a button except for the downward-pointing arrow on the right. When the user selects the text or the arrow, the list pops up and displays more selections. A popped-up list has a scroll bar on the right that works like the scroll bar of a list box. When a pop-up list is collapsed, the current selection appears in its display field. Pop-up lists do not allow multiple selection.

See “List Boxes and Pop-Up Lists” on page 380 for instructions on how to manage lists for list boxes and pop-up lists.

**label**                      Appears as a title to the left of the pop-up list. If specified, the `label` is left justified within the width of the `popup_list` tile.

**edit\_width**              Specifies the width of the text portion of the list in character-width units. It doesn't include the optional label on the left or the pop-up arrow (or scroll bar) on the right. If `edit_width` isn't specified or is zero, and the



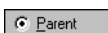
width of the tile isn't fixed, the box expands to fill the available space. Possible value is an integer or a real number. If `edit_width` is nonzero, then the box is right-justified within the space occupied by the tile. If it is necessary to stretch the tile for layout purposes, the PDB feature inserts white space between the `label` and the edit portion of the box.

`value`

A quoted string containing an integer (default: "0"). The integer is a zero-based index that indicates the currently selected item in the list (the one that is displayed when the list isn't popped up).

## radio\_button

```
: radio_button {
    action alignment fixed_height fixed_width
    height is_enabled is_tab_stop key label
    mnemonic value width
}
```



A radio button is one of a group of buttons composing a radio column or radio row. These work like the buttons on a car radio: only one can be selected at a time, and when one is pressed, any other button in the column (or row) that is on is turned off. An optional `label` appears to the right of the radio button. The PDB feature reports an error if you attempt to place a radio button outside a radio column or radio row.

`label`

The text displayed to the right of the radio button.

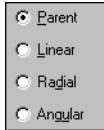
`value`

A quoted string (no default). If the `value` is "1", the `radio_button` is on; if it is "0", the `radio_button` is off; all other values are equivalent to "0".

If by some chance more than one `radio_button` in a radio cluster has `value = "1"`, only the last one is turned on. (This can happen in a DCL file. Once the dialog box starts, the PDB feature manages radio buttons and ensures that only one per cluster is turned on at a time.)

## radio\_column

```
: radio_column {  
    alignment children_alignment  
    children_fixed_height children_fixed_width  
    fixed_height fixed_width height label width  
}
```

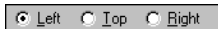


A radio column contains radio button tiles, only one of which can be selected at a time. Radio columns present the user with a fixed set of mutually exclusive choices. Radio columns, unlike ordinary columns, can be assigned an action.

**value** A quoted string containing the key of the currently selected radio button (the one whose value is "1").

## radio\_row

```
: radio_row {  
    alignment children_alignment  
    children_fixed_height children_fixed_width  
    fixed_height fixed_width height label width  
}
```



A radio row, like a radio column, contains radio button tiles, only one of which can be selected at a time. Radio rows can be assigned an action.

**value** A quoted string containing the key of the currently selected radio button (the one whose value is "1").

---

**NOTE** Radio rows are not as easy to use as radio columns, because the mouse has to travel farther. Use radio rows only if they specify two to four options, or if the labels are short.

---

## row

```
: row {  
    alignment children_alignment  
    children_fixed_height children_fixed_width  
    fixed_height fixed_width height label width  
}
```

A row is like a column, but its tiles are laid out horizontally instead of vertically, in the order they appear in the DCL file.

A row without a box has no additional attributes beyond the standard layout attributes.

## slider

```
: slider {  
    action alignment big_increment fixed_height  
    fixed_width height key label layout  
    max_value min_value mnemonic small_increment  
    value width  
}
```



A slider obtains a numeric value. The user can drag the slider's indicator to the left or right (or up or down) to obtain a value whose meaning depends on the application. This value is returned as a string containing a signed integer within a specified range (the integer is a 16-bit value, so the maximum range is -32,768 to 32,767). The application can scale this value as required.

**value**                      A quoted string that contains the current (integer) value of the slider (default: `min_value`).

## text

```
: text {  
    alignment fixed_height fixed_width height  
    is_bold key label value width  
}
```

A text tile displays a text string for titling or informational purposes.

Because most tiles have their own `label` attribute for titling purposes, you don't always need to use text tiles. But a text tile that you usually keep blank is a useful way to display feedback about user actions, error messages, or warnings.

Alert boxes and error tiles are discussed in “Dialog Box Exit Buttons and Error Tiles” on page 410 and “DCL Error Handling” on page 386.

If you intend the message to be static, specify it in the `label` attribute and don't specify a `width` or `value`. If you intend the message to change at run-time, specify it in the `value` attribute and assign a `width` long enough to contain any strings that you plan to assign the `value`. Once the dialog box is laid out, the size of its tiles can't change, so if you use `set_tile` to assign a string longer than the width, the displayed text is truncated.

**label**                      The displayed text. When a text tile is laid out, its width is the larger of either its `width` attribute, if that is specified in the DCL, or the width required by its `label` attribute, if specified. At least one of these attributes must be specified.

**value**                    Like `label`, the `value` attribute specifies a string to display in the text tile. However, it has no effect on the tile's layout.

## text\_part

```
: text_part {  
    label  
}
```

A text part is a text tile that is part of a larger piece of text. The margins of a `text_part` are suppressed, so it can be combined with other `text_parts` into a concatenation or paragraph tile.

The `text_part` tile is defined in the *base.dcl* file. See “paragraph” on page 419 for an example that uses `text_part`.

## toggle

```
: toggle {  
    action alignment fixed_height fixed_width  
    height is_enabled is_tab_stop label width  
}
```



A toggle controls a Boolean value ("0" or "1"). A toggle appears as a small box with an optional `label` to the right of the box. A check mark or *X* appears (or disappears) when the user selects the box. Toggles enable the user to view or change the state of on/off options. Toggles are also known as check boxes.

`label` The text displayed to the right of the toggle box.

`value` A quoted string containing an integer (default: "0") and specifying the initial state of the toggle. If the string is "0", the toggle box is blank (without a check mark). If it is "1", the box contains a check mark (or an *X*).

## spacer

```
: spacer {  
    alignment fixed_height fixed_width  
    height width  
}
```

A spacer is a blank tile. It is used only for layout purposes to affect the size and layout of adjacent tiles. To ensure consistency with other dialog boxes, use spacer tiles only in special cases, because the PDB feature handles spacing automatically. See "Adjusting the Layout of Dialog Boxes" on page 347.

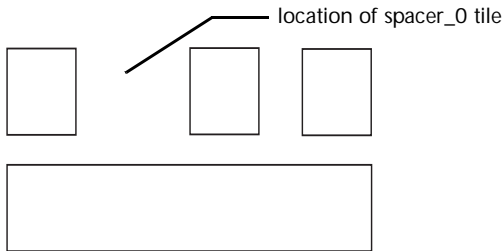
The `spacer` tile has no additional attributes beyond the standard layout attributes.

## spacer\_0

`spacer_0;`

A `spacer_0`, demonstrated in the following figure, is a spacer that normally has no width. However, it indicates a point in a tile group where you want space to be inserted, if the group has to be stretched during layout. If the `spacer_0` tiles in a group are assigned a positive width, all of them are assigned an equal share of the spacing.

The `spacer_0` tile is defined in the *base.dcl* file.

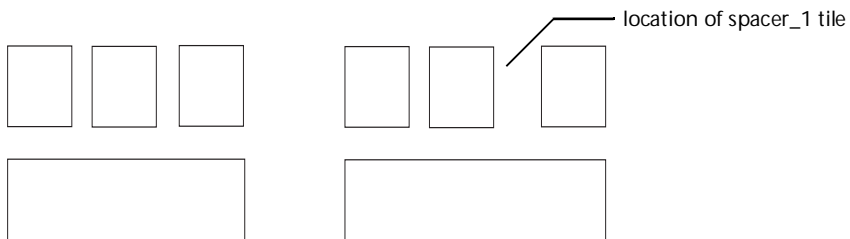


## spacer\_1

`spacer_1;`

The `spacer_1` tile, demonstrated in the following figure, is a spacer whose width and height both equal one. It is used for the smallest kind of spacer that will still be obvious to the user.

The `spacer_1` tile is defined in the *base.dcl* file.



# Programmable Dialog Box Function Synopsis

The programmable dialog box functions perform dialog box opening and closing, tile and attribute handling, list box and pop-up list handling, image tile handling, and application-specific data handling. This section lists each PDB function available in Visual LISP, grouping them by the type of task each performs. For detailed information on these functions, see the *AutoLISP Reference*.

## Dialog Box Opening and Closing Functions

Dialog box opening and closing functions	
Function name	Description
(done_dialog <i>[status]</i> )	Terminates a dialog box
(load_dialog <i>dclfile</i> )	Loads a DCL file
(new_dialog <i>dlgname dcl_id</i> <i>[action [screen-pt]]</i> )	Begins a new dialog box and displays it, and can also specify a default action
(start_dialog)	Displays a dialog box and begins accepting user input
(term_dialog)	Terminates all current dialog boxes as if the user cancels each of them
(unload_dialog <i>dcl_id</i> )	Unloads a DCL file

## Tile- and Attribute-Handling Functions

### Tile- and attribute-handling functions

Function name	Description
<b>(action_tile</b> <i>key action-expression</i> )	Assigns an action to evaluate when the user selects the specified tile in a dialog box
<b>(get_attr</b> <i>key attribute</i> )	Retrieves the DCL value of a dialog box attribute
<b>(get_tile</b> <i>key</i> )	Retrieves the current runtime value of a dialog box tile
<b>(mode_tile</b> <i>key mode</i> )	Sets the mode of a dialog box tile
<b>(set_tile</b> <i>key value</i> )	Sets the value of a dialog box tile

## List Box and Pop-Up List-Handling Functions

### List box and pop-up list-handling functions

Function name	Description
<b>(add_list</b> <i>string</i> )	Adds or modifies a string in the currently active dialog box list
<b>(end_list)</b>	Ends processing of the currently active dialog box list
<b>(start_list</b> <i>key [operation [index]]</i> )	Starts the processing of a list in the list box or in the pop-up list dialog box tile



## Image Tile-Handling Functions

Image tile-handling functions	
Function name	Description
<b>(dimx_tile <i>key</i>)</b> and <b>(dimy_tile <i>key</i>)</b>	Retrieves the dimensions of a tile in dialog box units
<b>(end_image)</b>	Ends creation of the currently active dialog box image
<b>(fill_image <i>x1 y1 wid hgt color</i>)</b>	Draws a filled rectangle in the currently active dialog box image tile
<b>(slide_image <i>x1 y1 wid hgt sldname</i>)</b>	Displays an AutoCAD slide in the currently active dialog box image tile
<b>(start_image <i>key</i>)</b>	Starts the creation of an image in the dialog box tile
<b>(vector_image <i>x1 y1 x2 y2 color</i>)</b>	Draws a vector in the currently active dialog box image

## Application-Specific Data-Handling Functions

Application-specific data-handling functions	
Function name	Description
<b>(client_data_tile <i>key clientdata</i>)</b>	Associates application-managed data with a dialog box tile



# **Part 4**

## **Appendixes**





# AutoLISP Function Synopsis

To find a function without knowing its name, use the listings in this appendix. The AutoLISP functions in this synopsis are organized into functional groups, and listed alphabetically within each function group. Each function is briefly described by its signature and a single sentence indicating the function's purpose.

## In This Appendix

# A

- Category Summary
- Basic Functions
- Utility Functions
- Selection Set, Object, and Symbol Table Functions
- Memory Management Functions
- Visual LISP Extensions to AutoLISP
- Reactor Functions
- VLX Namespace Functions
- Namespace Communication Functions
- Windows Registry Functions

# Category Summary

Functions in this synopsis are organized into the following categories:

- **Basic:** (application-handling, arithmetic, equality and conditional, error-handling, function-handling, list manipulation, string-handling, and symbol-handling functions)
- **Utility:** (conversion, device access, display control, file-handling, geometric, query and command, and user input functions)
- **Selection Set, Object, and Symbol Table:** (extended data-handling, object-handling, selection set manipulation, and symbol table-handling functions)
- **Memory Management**
- **Visual LISP Extensions to AutoLISP:** (collection manipulation, curve measurement, data conversion, dictionary-handling, object-handling, and property-handling functions)
- **Reactor**
- **VLX Namespace:** (function exposure, document namespace variable access, and error-handling functions)
- **Namespace Communication:** (blackboard-addressing and multi-document loading-functions)
- **Windows Registry Handling**

Note that programmable dialog box functions are listed in the “Programmable Dialog Box Reference” section of this manual.

Functions are grouped by data type and by the action they perform. Detailed information on each Visual LISP function is provided in the alphabetical listings in the *AutoLISP Reference*.

Note that any functions not described here or in other parts of the documentation are not officially supported and are subject to change in future releases.

## Basic Functions

Application-Handling Functions	(see page 436)
Arithmetic Functions	(see page 437)
Equality and Conditional Functions	(see page 439)
Error-Handling Functions	(see page 440)
Function-Handling Functions	(see page 441)
List Manipulation Functions	(see page 442)
String-Handling Functions	(see page 444)

Symbol-Handling Functions	(see page 445)
<b>Utility Functions</b>	
Conversion Functions	(see page 446)
Device Access Functions	(see page 447)
Display Control Functions	(see page 448)
File-Handling Functions	(see page 449)
Geometric Functions	(see page 450)
Query and Command Functions	(see page 451)
User Input Functions	(see page 452)
<b>Selection Set, Object, and Symbol Table Functions</b>	
Extended Data-Handling Functions	(see page 453)
Object-Handling Functions	(see page 453)
Selection Set Manipulation Functions	(see page 455)
Symbol Table and Dictionary-Handling Functions	(see page 455)
<b>Memory Management Functions</b>	(see page 457)
<b>Visual LISP AutoLISP Extensions</b>	
ActiveX Collection Manipulation Functions	(see page 458)
ActiveX Data Conversion Functions	(see page 458)
ActiveX Method Invocation Functions	(see page 459)
ActiveX Object-Handling Functions	(see page 460)
ActiveX Property-Handling Functions	(see page 460)
Curve Measurement Functions	(see page 461)
Dictionary Functions	(see page 462)
Object-Handling Functions	(see page 463)
<b>Reactor Functions</b>	(see page 464)
<b>VLX Namespace Functions</b>	(see page 467)
<b>Namespace Communication Functions</b>	(see page 468)
<b>Windows Registry Functions</b>	(see page 469)

# Basic Functions

The basic functions consist of the arithmetic, string-handling, equality and conditional, list manipulation, symbol-handling, function-handling, error-handling, and application-handling functions.

## Application-Handling Functions

Application-handling functions	
Function	Description
(arx)	Returns a list of the currently loaded ObjectARX applications
(arxload <i>application</i> [ <i>onfailure</i> ])	Loads an ObjectARX application
(arxunload <i>application</i> [ <i>onfailure</i> ])	Unloads an ObjectARX application
(autoarxload <i>filename cmdlist</i> )	Predefines command names to load an associated ObjectARX file
(autoload <i>filename cmdlist</i> )	Predefines command names to load an associated AutoLISP file
(initdia [ <i>dialogflag</i> ])	Forces the display of the next command's dialog box
(load <i>filename</i> [ <i>onfailure</i> ])	Evaluates the AutoLISP expressions in a file
(startapp <i>appcmd file</i> )	Starts a Windows application
(vl-vbaload " <i>filename</i> ")	Loads a Visual Basic project
(vl-vbarun " <i>macroname</i> ")	Runs a Visual Basic macro
(vlax-add-cmd " <i>global-name</i> " <i>'func-sym</i> [" <i>local-name</i> " <i>cmd-flags</i> ])	Adds commands to AutoCAD's built-in command set <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>



# Arithmetic Functions

Arithmetic functions	
Function	Description
(+ [ <i>number number</i> ] ...)	Returns the sum of all numbers
(− [ <i>number number</i> ] ...)	Subtracts the second and following numbers from the first and returns the difference
(* [ <i>number number</i> ] ...)	Returns the product of all numbers
(/ [ <i>number number</i> ] ...)	Divides the first number by the product of the remaining numbers and returns the quotient
(~ <i>int</i> )	Returns the bitwise NOT (1's complement) of the argument
(1+ <i>number</i> )	Returns the argument increased by 1 (incremented)
(1− <i>number</i> )	Returns the argument reduced by 1 (decremented)
(abs <i>number</i> )	Returns the absolute value of the argument
(atan <i>num1</i> [ <i>num2</i> ])	Returns the arctangent of a number in radians
(cos <i>ang</i> )	Returns the cosine of an angle expressed in radians
(exp <i>number</i> )	Returns the constant e (a real) raised to a specified power (the natural antilog)
(expt <i>base power</i> )	Returns a number raised to a specified power
(fix <i>number</i> )	Returns the conversion of a real into the nearest smaller integer
(float <i>number</i> )	Returns the conversion of a number into a real
(gcd <i>int1 int2</i> )	Returns the greatest common denominator of two integers
(log <i>number</i> )	Returns the natural log of a number as a real
(logand [ <i>int int</i> ] ...)	Returns the result of the logical bitwise AND of a list of integers
(logior [ <i>int int</i> ] ...)	Returns the result of the logical bitwise inclusive OR of a list of integers

### Arithmetic functions (*continued*)

Function	Description
(lsh [ <i>int numbits</i> ])	Returns the logical bitwise shift of an integer by a specified number of bits
(max [ <i>number number ...</i> ])	Returns the largest of the numbers given
(min [ <i>number number ...</i> ])	Returns the smallest of the numbers given
(minusp <i>number</i> )	Verifies that a number is negative
(rem [ <i>num1 num2 ...</i> ])	Divides the first number by the second, and returns the remainder
(sin <i>ang</i> )	Returns the sine of an angle as a real expressed in radians
(sqrt <i>number</i> )	Returns the square root of a number as a real
(zerop <i>number</i> )	Verifies that a number evaluates to zero

# Equality and Conditional Functions

Equality and conditional functions	
Function	Description
(= <i>numstr</i> [ <i>numstr</i> ] ...)	Returns T if all arguments are numerically equal, and returns nil otherwise
(/ = <i>numstr</i> [ <i>numstr</i> ] ...)	Returns T if the arguments are not numerically equal, and nil if the arguments are numerically equal
(< <i>numstr</i> [ <i>numstr</i> ] ...)	Returns T if each argument is numerically less than the argument to its right, and returns nil otherwise
(<= <i>numstr</i> [ <i>numstr</i> ] ...)	Returns T if each argument is numerically less than or equal to the argument to its right, and returns nil otherwise
(> <i>numstr</i> [ <i>numstr</i> ] ...)	Returns T if each argument is numerically greater than the argument to its right, and returns nil otherwise
(>= <i>numstr</i> [ <i>numstr</i> ] ...)	Returns T if each argument is numerically greater than or equal to the argument to its right, and returns nil otherwise
(and [ <i>expr</i> ] ...)	Returns the logical AND of a list of expressions
(Boole <i>func</i> <i>int1</i> [ <i>int2</i> ] ...)	Serves as a general bitwise Boolean function
(cond [( <i>test result</i> ...) ...])	Serves as the primary conditional function for AutoLISP
(eq <i>expr1</i> <i>expr2</i> )	Determines whether two expressions are identical
(equal <i>expr1</i> <i>expr2</i> [ <i>fuzz</i> ])	Determines whether two expressions are equal
(if <i>testexpr</i> <i>thenexpr</i> [ <i>elseexpr</i> ])	Conditionally evaluates expressions
(or [ <i>expr</i> ] ...)	Returns the logical OR of a list of expressions
(repeat <i>int</i> [ <i>expr</i> ] ...)	Evaluates each expression a specified number of times, and returns the value of the last expression
(while <i>testexpr</i> [ <i>expr</i> ] ...)	Evaluates a test expression, and if it is not nil, evaluates other expressions; repeats this process until the test expression evaluates to nil

## Error-Handling Functions

Error-handling functions	
Function	Description
(alert <i>string</i> )	Displays an alert dialog box with the error or warning message passed as a string
(*error* <i>string</i> )	A user-definable error-handling function
(exit)	Forces the current application to quit
(quit)	Forces the current application to quit
(vl-catch-all-apply ' <i>function list</i> )	Passes a list of arguments to a specified function and traps any exceptions
(vl-catch-all-error-message <i>error-obj</i> )	Returns a string from an error object
(vl-catch-all-error-p <i>arg</i> )	Determines whether an argument is an error object returned from <b>vl-catch-all-apply</b>

# Function-Handling Functions

Function-handling functions	
Function	Description
(apply <i>function lst</i> )	Passes a list of arguments to a specified function
(defun <i>sym</i> ([ <i>arguments</i> ] [/variables...]) <i>expr</i> ...)	Defines a function
(defun-q <i>sym</i> ([ <i>arguments</i> ] [/variables...]) <i>expr</i> ...)	Defines a function as a list (intended for backward-compatibility only)
(defun-q-list-ref ' <i>function</i> )	Displays the list structure of a function defined with <b>defun-q</b>
(defun-q-list-set ' <i>sym list</i> )	Defines a function as a list (intended for backward-compatibility only)
(eval <i>expr</i> )	Returns the result of evaluating an AutoLISP expression
(lambda <i>arguments expr</i> ...)	Defines an anonymous function
(progn [ <i>expr</i> ] ...)	Evaluates each expression sequentially, and returns the value of the last expression
(trace <i>function</i> ...)	Aids in AutoLISP debugging
(untrace <i>function</i> ...)	Clears the trace flag for the specified functions

# List Manipulation Functions

List manipulation functions	
Function	Description
(acad_strlsort <i>lst</i> )	Sorts a list of strings by alphabetical order
(append <i>lst</i> ...)	Takes any number of lists and runs them together as one list
(assoc <i>item alist</i> )	Searches an association list for an element and returns that association list entry
(car <i>lst</i> )	Returns the first element of a list
(cdr <i>lst</i> )	Returns the specified list, except for the first element of the list
(cons <i>new-first-element lst</i> )	The basic list constructor
(foreach <i>name lst [expr ...]</i> )	Evaluates expressions for all members of a list
(last <i>lst</i> )	Returns the last element in a list
(length <i>lst</i> )	Returns an integer indicating the number of elements in a list
(list [ <i>expr</i> ...])	Takes any number of expressions and combines them into one list
(listp <i>item</i> )	Verifies that an item is a list
(mapcar <i>function list1 ... listn</i> )	Returns a list of the result of executing a function with the individual elements of a list or lists supplied as arguments to the function
(member <i>expr lst</i> )	Searches a list for an occurrence of an expression and returns the remainder of the list, starting with the first occurrence of the expression
(nth <i>n lst</i> )	Returns the <i>n</i> th element of a list
(reverse <i>lst</i> )	Returns a list with its elements reversed
(subst <i>newitem olditem lst</i> )	Searches a list for an old item and returns a copy of the list with a new item substituted in place of every occurrence of the old item
(vl-consp <i>list-variable</i> )	Determines whether or not a list is nil

### List manipulation functions (*continued*)

Function	Description
(vl-every <i>predicate-function list</i> [ <i>more-lists</i> ]...)	Checks whether the predicate is true for every element combination
(vl-list* <i>object</i> [ <i>more-objects</i> ]...)	Constructs and returns a list
(vl-list->string <i>char-codes-list</i> )	Combines the characters associated with a list of integers into a string
(vl-list-length <i>list-or-cons-object</i> )	Calculates list length of a true list
(vl-member-if <i>predicate-function list</i> )	Determines whether the predicate is true for one of the list members
(vl-member-if-not <i>predicate-function list</i> )	Determines whether the predicate is nil for one of the list members
(vl-position <i>symbol list</i> )	Returns the index of the specified list item
(vl-remove <i>element-to-remove list</i> )	Removes elements from a list
(vl-remove-if <i>predicate-function list</i> )	Returns all elements of the supplied list that fail the test function
(vl-remove-if-not <i>predicate-function list</i> )	Returns all elements of the supplied list that pass the test function
(vl-some <i>predicate-function list</i> [ <i>more-lists</i> ]...)	Checks whether the predicate is not nil for one element combination
(vl-sort <i>list</i> less?-function)	Sorts the elements in a list according to a given compare function
(vl-sort-i <i>list</i> less?-function)	Sorts the elements in a list according to a given compare function, and returns the element index numbers
(vl-string->list <i>string</i> )	Converts a string into a list of character codes

# String-Handling Functions

String-handling functions	
Function	Description
(read <i>[string]</i> )	Returns the first list or atom obtained from a string
(strcase <i>string [which]</i> )	Returns a string where all alphabetic characters have been converted to uppercase or lowercase
(strcat <i>[string1 [string2] ...]</i> )	Returns a string that is the concatenation of multiple strings
(strlen <i>[string] ...]</i> )	Returns an integer that is the number of characters in a string
(subst <i>string start [length]</i> )	Returns a substring of a string
(vl-prin1-to-string <i>object</i> )	Returns the string representation of any LISP object as if it were output by the <b>prin1</b> function
(vl-princ-to-string <i>object</i> )	Returns the string representation of any LISP object as if it were output by the <b>princ</b> function
(vl-string->list <i>string</i> )	Converts a string into a list of character codes
(vl-string-elt <i>string position</i> )	Returns the ASCII representation of the character at a specified position in a string
(vl-string-left-trim <i>character-set string</i> )	Removes the specified characters from the beginning of a string
(vl-string-mismatch <i>str1 str2 [pos1 pos2 ignore-case-p]</i> )	Returns the length of the longest common prefix for two strings, starting at specified positions
(vl-string-position <i>char-code str [start-pos [from-end-p]]</i> )	Looks for a character with the specified ASCII code in a string
(vl-string-right-trim <i>character-set string</i> )	Removes the specified characters from the end of a string
(vl-string-search <i>pattern string [start-pos]</i> )	Searches for the specified pattern in a string
(vl-string-subst <i>new-str pattern string [start-pos]</i> )	Substitutes one string for another, within a string
(vl-string-translate <i>source-set dest-set str</i> )	Replaces characters in a string with a specified set of characters



### String-handling functions (*continued*)

Function	Description
(vl-string-trim <i>char-set str</i> )	Removes the specified characters from the beginning and end of a string
(wcmatch <i>string pattern</i> )	Performs a wild-card pattern match on a string

## Symbol-Handling Functions

### Symbol-handling functions

Function	Description
(atom <i>item</i> )	Verifies that an item is an atom
(atoms-family <i>format [symlist]</i> )	Returns a list of the currently defined symbols
(boundp <i>sym</i> )	Verifies whether a value is bound to a symbol
(not <i>item</i> )	Verifies that an item evaluates to nil
(null <i>item</i> )	Verifies that an item is bound to nil
(numberp <i>item</i> )	Verifies that an item is a real or an integer
(quote <i>expr</i> )	Returns an expression without evaluating it
(set <i>sym expr</i> )	Sets the value of a quoted symbol name to an expression
(setq <i>sym1 expr1</i> <i>[sym2 expr2] ...</i> )	Sets the value of a symbol or symbols to associated expressions
(type <i>item</i> )	Returns the type of a specified item
(vl-symbol-name <i>symbol</i> )	Returns a string containing the name of a symbol
(vl-symbol-value <i>symbol</i> )	Returns the current value bound to a symbol
(vl-symbolp <i>object</i> )	Identifies whether or not a specified object is a symbol

# Utility Functions

The utility functions consist of query and command, display control, user input, geometric, conversion, file-handling, and device access functions.

## Conversion Functions

Conversion functions	
Function	Description
(angtof <i>string</i> [ <i>mode</i> ])	Converts a string representing an angle into a real (floating-point) value in radians
(angtos <i>angle</i> [ <i>mode</i> [ <i>precision</i> ]])	Converts an angular value in radians into a string
(ascii <i>string</i> )	Returns the conversion of the first character of a string into its ASCII character code (an integer)
(atof <i>string</i> )	Returns the conversion of a string into a real
(atoi <i>string</i> )	Returns the conversion of a string into an integer
(chr <i>integer</i> )	Returns the conversion of an integer representing an ASCII character code into a single-character string
(cvunit <i>value from to</i> )	Converts a value from one unit of measurement to another
(distof <i>string</i> [ <i>mode</i> ])	Converts a string that represents a real (floating-point) value into a real value
(itoa <i>int</i> )	Returns the conversion of an integer into a string
(rtos <i>number</i> [ <i>mode</i> [ <i>precision</i> ]])	Converts a number into a string
(trans <i>pt from to</i> [ <i>disp</i> ])	Translates a point (or a displacement) from one coordinate system to another

# Device Access Functions

Device access functions	
Function	Description
(gread [ <i>track</i> ] [ <i>allkeys</i> ] [ <i>curtype</i> ]))	Reads values from any of the AutoCAD input devices
(tablet <i>code</i> [ <i>row1 row2 row3 direction</i> ]))	Retrieves and sets digitizer (tablet) calibrations

# Display Control Functions

Display control functions	
Function	Description
(graphscr)	Displays the AutoCAD graphics screen
(grdraw <i>from to color [highlight]</i> )	Draws a vector between two points, in the current viewport
(grtext <i>[box text [highlight]]</i> )	Writes text to the status line or to screen menu areas
(grvecs <i>vlist [trans]</i> )	Draws multiple vectors on the graphics screen
(menucmd <i>string</i> )	Issues menu commands, or sets and retrieves menu item status
(menugroup <i>groupname</i> )	Verifies that a menu group is loaded
(prin1 <i>[expr [file-desc]]</i> )	Prints an expression to the command line or writes an expression to an open file
(princ <i>[expr [file-desc]]</i> )	Prints an expression to the command line, or writes an expression to an open file
(print <i>[expr [file-desc]]</i> )	Prints an expression to the command line, or writes an expression to an open file
(prompt <i>msg</i> )	Displays a string on your screen's prompt area
(redraw <i>[ename [mode]]</i> )	Redraws the current viewport or a specified object (entity) in the current viewport
(terpri)	Prints a newline to the Command line
(textpage)	Switches from the graphics screen to the text screen
(textscr)	Switches from the graphics screen to the text screen (like the AutoCAD Flip Screen function key)
(vports)	Returns a list of viewport descriptors for the current viewport configuration

# File-Handling Functions

File-handling functions	
Function	Description
(close <i>file-desc</i> )	Closes an open file
(findfile <i>filename</i> )	Searches the AutoCAD library path for the specified file
(open <i>filename mode</i> )	Opens a file for access by the AutoLISP I/O functions
(read-char [ <i>file-desc</i> ])	Returns the decimal ASCII code representing the character read from the keyboard input buffer or from an open file
(read-line [ <i>file-desc</i> ])	Reads a string from the keyboard or from an open file
(vl-directory-files [ <i>directory pattern</i> [ <i>Lists all files in a given directory directories</i> ]])	
(vl-file-copy " <i>source-filename</i> " " <i>destination-filename</i> " [ <i>append?</i> ])	Copies or appends the contents of one file to another file
(vl-file-delete " <i>filename</i> ")	Deletes a file
(vl-file-directory-p " <i>filename</i> ")	Determines if a file name refers to a directory
(vl-file-rename " <i>old-filename</i> " " <i>new-filename</i> ")	Renames a file
(vl-file-size " <i>filename</i> ")	Determines the size of a file, in bytes
(vl-file-systime " <i>filename</i> ")	Returns last modification time of the specified file
(vl-filename-base " <i>filename</i> ")	Returns the name of a file, after stripping out the directory path and extension
(vl-filename-directory " <i>filename</i> ")	Returns the directory path of a file, after stripping out the name and extension
(vl-filename-extension " <i>filename</i> ")	Returns the extension from a file name, after stripping out the rest of the name
(vl-filename-mktemp [ <i>"pattern"</i> " <i>directory</i> " " <i>extension</i> "])	Calculates a unique file name to be used for a temporary file
(write-char <i>num</i> [ <i>file-desc</i> ])	Writes one character to the screen or to an open file
(write-line <i>string</i> [ <i>file-desc</i> ])	Writes a string to the screen or to an open file

## Geometric Functions

Geometric functions	
Function	Description
(angle <i>pt1 pt2</i> )	Returns an angle in radians of a line defined by two endpoints
(distance <i>pt1 pt2</i> )	Returns the 3D distance between two points
(inters <i>pt1 pt2 pt3 pt4 [onseg]</i> )	Finds the intersection of two lines
(osnap <i>pt mode</i> )	Returns a 3D point that is the result of applying an Object Snap mode to a specified point
(polar <i>pt ang dist</i> )	Returns the UCS 3D point at a specified angle and distance from a point
(textbox <i>elist</i> )	Measures a specified text object, and returns the diagonal coordinates of a box that encloses the text

# Query and Command Functions

Query and command functions	
Function	Description
(acad_colordlg <i>colornum</i> [ <i>flag</i> ])	Displays the standard AutoCAD Color Selection dialog box
(acad_helpdlg <i>helpfile</i> <i>topic</i> )	Invokes the Help facility (obsolete)
(command [ <i>arguments</i> ] ...)	Executes an AutoCAD command
(getcfg <i>cfgname</i> )	Retrieves application data from the AppData section of the <i>acad.cfg</i> file
(getcname <i>cname</i> )	Retrieves the localized or English name of an AutoCAD command
(getenv "variable-name")	Returns the string value assigned to a system environment variable
(getvar <i>varname</i> )	Retrieves the value of an AutoCAD system variable
(help [ <i>helpfile</i> [ <i>topic</i> [ <i>command</i> ]]])	Invokes the Help facility
(setcfg <i>cfgname</i> <i>cfgval</i> )	Writes application data to the AppData section of the <i>acad.cfg</i> file
(setenv "varname" "value")	Sets a system environment variable to a specified value
(setfunhelp "c:fname" ["helpfile" ["topic" ["command"]]])	Registers a user-defined command with the Help facility so the appropriate help file and topic are called when the user requests help on that command
(setvar <i>varname</i> <i>value</i> )	Sets an AutoCAD system variable to a specified value
(ver)	Returns a string that contains the current AutoLISP version number
(vl-cmdf [ <i>arguments</i> ] ...)	Executes an AutoCAD command after evaluating <i>arguments</i>
(vlax-add-cmd <i>global-name</i> <i>func-sym</i> [ <i>local-name</i> <i>cmd-flags</i> ])	Adds commands to a group <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-remove-cmd <i>global-name</i> )	Removes a single command or command group <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>

# User Input Functions

User input functions	
Function	Description
(entsel <i>[msg]</i> )	Prompts the user to select a single object (entity) by specifying a point
(getangle <i>[pt] [msg]</i> )	Pauses for user input of an angle, and returns that angle in radians
(getcorner <i>pt [msg]</i> )	Pauses for user input of a rectangle's second corner
(getdist <i>[pt] [msg]</i> )	Pauses for user input of a distance
(getfiled <i>title default ext flags</i> )	Prompts the user for a file name with the standard AutoCAD file dialog box, and returns that file name
(getint <i>[msg]</i> )	Pauses for user input of an integer, and returns that integer
(getkeyword <i>[msg]</i> )	Pauses for user input of a keyword, and returns that keyword
(getorient <i>[pt] [msg]</i> )	Pauses for user input of an angle, and returns that angle in radians
(getpoint <i>[pt] [msg]</i> )	Pauses for user input of a point, and returns that point
(getreal <i>[msg]</i> )	Pauses for user input of a real number, and returns that real number
(getstring <i>[cr] [msg]</i> )	Pauses for user input of a string, and returns that string
(initget <i>[bits] [string]</i> )	Establishes keywords for use by the next user input function call
(nentsel <i>[msg]</i> )	Prompts the user to select an object (entity) by specifying a point, and provides access to the definition data contained within a complex object
(nentselp <i>[msg] [pt]</i> )	Provides similar functionality to that of the nentsel function without the need for user input



# Selection Set, Object, and Symbol Table Functions

The selection set, object, and symbol table functions consist of selection set manipulation, object-handling, extended data-handling, and symbol table-handling functions.

## Extended Data-Handling Functions

Extended data-handling functions	
Function	Description
(regapp <i>application</i> )	Registers an application name with the current AutoCAD drawing in preparation for using extended object data
(xdroom <i>ename</i> )	Returns the amount of extended data (xdata) space that is available for an object (entity)
(xdsize <i>lst</i> )	Returns the size (in bytes) that a list occupies when it is linked to an object (entity) as extended data

## Object-Handling Functions

Object-handling functions	
Function	Description
(entdel <i>ename</i> )	Deletes objects (entities) or restores previously deleted objects
(entget <i>ename [applist]</i> )	Retrieves an object's (entity's) definition data
(entlast)	Returns the name of the last nondeleted main object (entity) in the drawing
(entmake <i>elist</i> )	Creates a new entity (graphical object) in the drawing
(entmakex <i>elist</i> )	Makes a new object or entity, gives it a handle and entity name (but, does not assign an owner), and then returns the new entity name
(entmod <i>elist</i> )	Modifies the definition data of an object (entity)

## Object-handling functions (*continued*)

Function	Description
(entnext <i>[ename]</i> )	Returns the name of the next object (entity) in the drawing
(entupd <i>ename</i> )	Updates the screen image of an object (entity)
(handent <i>handle</i> )	Returns an object (entity) name based on its handle
(vlax-dump-object <i>obj</i> )	Lists an object's methods and properties <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-erased-p <i>obj</i> )	Determines whether an object was erased <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-get-acad-object)	Retrieves the top-level AutoCAD application object for the current AutoCAD session <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-method-applicable-p <i>obj method</i> )	Determines whether an object supports a particular method <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-object-released-p <i>obj</i> )	Determines whether an object has been released <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-read-enabled-p <i>obj</i> )	Determines whether an object can be read <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-release-object <i>obj</i> )	Releases a drawing object <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-typeinfo-available-p <i>obj</i> )	Determines whether TypeLib information is present for the specified type of object <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-write-enabled-p <i>obj</i> )	Determines whether an AutoCAD drawing object can be modified <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>

## Selection Set Manipulation Functions

Selection set manipulation functions	
Function	Description
(ssadd <i>[ename [ss]]</i> )	Adds an object (entity) to a selection set, or creates a new selection set
(ssdel <i>ename ss</i> )	Deletes an object (entity) from a selection set
(ssget <i>[mode] [pt1 [pt2]] [pt-list] [filter-list]</i> )	Prompts the user to select objects (entities), and returns a selection set
(ssgetfirst)	Determines which objects are selected and gripped
(sslength <i>ss</i> )	Returns an integer containing the number of objects (entities) in a selection set
(ssmemb <i>ename ss</i> )	Tests whether an object (entity) is a member of a selection set
(ssname <i>ss index</i> )	Returns the object (entity) name of the indexed element of a selection set
(ssnamex <i>ss index</i> )	Retrieves information about how a selection set was created
(sssetfirst <i>gripset [pickset]</i> )	Sets which objects are selected and gripped

## Symbol Table and Dictionary-Handling Functions

Symbol table and dictionary-handling functions	
Function	Description
(dictadd <i>ename symbol newobj</i> )	Adds a non-graphical object to the specified dictionary
(dictnext <i>ename symbol [rewind]</i> )	Finds the next item in a dictionary
(dictremove <i>ename symbol</i> )	Removes an entry from the specified dictionary
(dictrename <i>ename oldsym newsym</i> )	Renames a dictionary entry
(dictsearch <i>ename symbol [setnext]</i> )	Searches a dictionary for an item

## Symbol table and dictionary-handling functions (*continued*)

Function	Description
(layoutlist)	Returns a list of all paper space layouts in the current drawing
(namedobjdict)	Returns the entity name of the current drawing's named object dictionary, which is the root of all non-graphical objects in the drawing
(setview <i>view_description</i> <i>[vport_id]</i> )	Establishes a view for a specified viewport
(snvalid <i>sym_name</i> )	Checks the symbol table name for valid characters
(tblnext <i>table-name</i> [ <i>rewind</i> ])	Finds the next item in a symbol table
(tblobjname <i>table-name symbol</i> )	Returns the entity name of a specified symbol table entry
(tblsearch <i>table-name symbol</i> [ <i>setnext</i> ])	Searches a symbol table for a symbol name
(vlax-ldata-delete <i>dict key</i> )	Erases LISP data from a drawing dictionary <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-ldata-get <i>dict key</i> [ <i>default-data</i> ])	Retrieves LISP data from a drawing dictionary <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-ldata-list <i>dict</i> )	Lists LISP data in a drawing dictionary <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-ldata-put <i>dict key data</i> )	Stores LISP data in a drawing dictionary <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>
(vlax-ldata-test <i>data</i> )	Determines whether data can be saved over a session boundary <b>NOTE</b> VLISP extension: requires <b>vl-load-com</b>

# Memory Management Functions

Memory management functions	
Function	Description
(alloc <i>int</i> )	Sets the segment size to a given number of nodes
(expand <i>number</i> )	Allocates node space by requesting a specified number of segments
(gc)	Forces a garbage collection, which frees up unused memory
(mem)	Displays the current state of memory in AutoLISP

## Visual LISP Extensions to AutoLISP

The extended AutoLISP functions provided with VLISP consist of curve measurement, data conversion, object-handling, property-handling, collection manipulation, and dictionary-handling functions. The function names are prefixed with *vlax-*. These functions are in addition to the ActiveX methods provided through *vla-\** functions.

**NOTE** Before you can use the AutoLISP extensions, you must issue the following command:

(vl -l oad-com)

The vl -l oad-com function also initializes ActiveX support for AutoLISP.

# ActiveX Collection Manipulation Functions

## Collection manipulation functions

Function	Description
(vlax-for <i>symbol collection</i> [ <i>expression1</i> [ <i>expression2</i> ...]])	Iterates through a collection of objects, evaluating each expression (VLISP Function)
(vlax-map-collection <i>obj function</i> )	Applies a function to all objects in a collection

# ActiveX Data Conversion Functions

## Data conversion functions

Function	Description
(vlax-3D-point <i>list</i> ) (vlax-3D-point <i>x y [z]</i> )	Creates an ActiveX-compatible 3D point structure
(vlax-ename->vla-object <i>entname</i> )	Transforms entity to VLA-object
(vlax-make-safearray <i>type</i> ' <i>l-bound . u-bound</i> ) [ ' <i>l-bound . u-bound</i> ...)]	Creates a safearray
(vlax-make-variant <i>value type</i> )	Creates a variant data type
(vlax-safearray-fill <i>var</i> ' <i>element-values</i> )	Stores elements in a safearray
(vlax-safearray-get-dim <i>var</i> )	Returns the number of dimensions in a safearray object
(vlax-safearray-get-element <i>var element</i> )	Returns an element from an array
(vlax-safearray-get-l-bound <i>var dim</i> )	Returns the lower boundary (starting index) of a dimension of an array
(vlax-safearray-get-u-bound <i>var dim</i> )	Returns the upper boundary (end index) of a dimension of an array
(vlax-safearray-put-element <i>var element value</i> )	Adds or updates an element in an array
(vlax-safearray-type <i>var</i> )	Returns the data type of a safearray

### Data conversion functions (*continued*)

Function	Description
(vlax-safearray->list <i>var</i> )	Returns the elements of a safearray in list form
(vlax-tmatrix <i>list</i> )	Returns a suitable representation for a 4x4 transformation matrix to be used in VLA methods
(vlax-variant-change-type <i>var type</i> )	Returns the value of a variant after changing it from one data type to another
(vlax-variant-type <i>var</i> )	Returns the data type of a variant
(vlax-variant-value <i>var</i> )	Returns the value of a variant
(vlax-vla-object->ename <i>obj</i> )	Transforms a VLA-object to an AutoLISP entity

## ActiveX Method Invocation Functions

### Method invocation functions

Function	Description
(vlax-invoke-method <i>obj method list</i> )	Calls the specified method of an object
(vlax-method-applicable-p <i>obj method</i> )	Determines if an object supports a particular method

## ActiveX Object-Handling Functions

ActiveX Object-handling functions	
Function	Description
(vlax-dump-object <i>obj</i> )	Lists an object's methods and properties
(vlax-erased-p <i>obj</i> )	Determines whether an object was erased
(vlax-get-acad-object)	Retrieves the top-level AutoCAD application object for the current AutoCAD session
(vlax-method-applicable-p <i>obj method</i> )	Determines if an object supports a particular method
(vlax-object-released-p <i>obj</i> )	Determines if an object has been released
(vlax-read-enabled-p <i>obj</i> )	Determines whether an object can be read
(vlax-release-object <i>obj</i> )	Releases a drawing object
(vlax-typeinfo-available-p <i>obj</i> )	Determines whether TypeLib information is present for the specified type of object
(vlax-write-enabled-p <i>obj</i> )	Determines whether an AutoCAD drawing object can be modified

## ActiveX Property-Handling Functions

Property-handling functions	
Function	Description
(vlax-get-property <i>obj property</i> )	Low-level property get function. May be used for custom ActiveX object
(vlax-property-available-p <i>obj prop [T]</i> )	Determines whether an object has a specified property
(vlax-put-property <i>obj property arg</i> )	Low-level property set function



## Curve Measurement Functions

Curve measurement functions	
Function	Description
(vlax-curve-getArea <i>curve-obj</i> )	Returns the area inside the curve
(vlax-curve-getDistAtParam <i>curve-obj param</i> )	Returns the length of the curve's segment from the curve's beginning to the specified point
(vlax-curve-getDistAtPoint <i>curve-obj point</i> )	Returns the length of the curve's segment between the curve's start point and the specified point
(vlax-curve-getEndParam <i>curve-obj</i> )	Returns the parameter of the endpoint of the curve
(vlax-curve-getEndPoint <i>curve-obj</i> )	Returns the endpoint (in WCS coordinates) of the curve
(vlax-curve-getParamAtDist <i>curve-obj param</i> )	Returns the distance along the curve from the beginning of the curve to the location of the specified parameter
(vlax-curve-getParamAtPoint <i>curve-obj point</i> )	Returns the parameter of the curve at the point
(vlax-curve-getPointAtDist <i>curve-obj dist</i> )	Returns the point (in WCS coordinates) along a curve at the distance specified by the user
(vlax-curve-getPointAtParam <i>curve-obj param</i> )	Determines the point on the curve that corresponds to the <i>param</i> parameter and returns the point
(vlax-curve-getStartParam <i>curve-obj</i> )	Returns the start parameter on the curve
(vlax-curve-getStartPoint <i>curve-obj</i> )	Returns the start point (in WCS coordinates) of the curve
(vlax-curve-isClosed <i>curve-obj</i> )	Determines if the specified curve is closed (i.e., start point is same as endpoint)
(vlax-curve-isPeriodic <i>curve-obj</i> )	Determines if the specified curve has an infinite range in both directions and there is a period value $dT$ , such that there is a point on curve at $(u + dT) = \text{point on curve } (u)$ , for any parameter $u$
(vlax-curve-isPlanar <i>curve-obj</i> )	Determines if there is a plane that contains the curve

### Curve measurement functions (*continued*)

Function	Description
(vlax-curve-getClosestPointTo <i>curve-obj givenPnt [extend]</i> )	Returns the point (in WCS coordinates) on a curve that is nearest to the specified point
(vlax-curve- getClosestPointToProjection <i>curve-obj givenPnt normal</i> [ <i>extend</i> ])	Returns the point (in WCS coordinates) on a curve that is nearest to the specified point
(vlax-curve-getFirstDeriv <i>curve-obj param</i> )	Returns the first derivative (in WCS coordinates) of a curve at the specified location
(vlax-curve-getSecondDeriv <i>curve-obj param</i> )	Returns the second derivative (in WCS coordinates) of a curve at the specified location

## Dictionary Functions

### Dictionary functions

Function	Description
(vlax-ldata-delete <i>dict key</i> )	Erases LISP data from a drawing dictionary
(vlax-ldata-get <i>dict key</i> [ <i>default-data</i> ])	Retrieves LISP data from a drawing dictionary
(vlax-ldata-list <i>dict</i> )	Lists LISP data in a drawing dictionary
(vlax-ldata-put <i>dict key data</i> )	Stores LISP data in a drawing dictionary
(vlax-ldata-test <i>data</i> )	Determines whether data can be saved over a session boundary

## Object-Handling Functions

Object-handling functions	
Function	Description
(vlax-create-object " <i>prog-id</i> ")	Creates a new instance of an ActiveX object
(vlax-dump-object <i>obj</i> )	Lists an object's methods and properties
(vlax-erased-p <i>obj</i> )	Determines whether an object was erased
(vlax-get-acad-object)	Retrieves the top-level AutoCAD application object for the current AutoCAD session
(vlax-get-object " <i>prog-id</i> ")	Returns a running instance of an ActiveX object
(vlax-get-or-create-object " <i>prog-id</i> ")	Returns a running instance of an ActiveX object, if one exists, otherwise starts a new instance of the object
(vlax-import-type-library :tlb-filename <i>filename</i> [ :methods-prefix <i>mprefix</i> :properties-prefix <i>pprefix</i> :constants-prefix <i>cprefix</i> ])	Imports information from a type library
(vlax-method-applicable-p <i>obj</i> <i>method</i> )	Determines whether an object supports a particular method
(vlax-object-released-p <i>obj</i> )	Determines whether an object has been released
(vlax-read-enabled-p <i>obj</i> )	Determines whether an object can be read
(vlax-release-object <i>obj</i> )	Releases a drawing object
(vlax-typeinfo-available-p <i>obj</i> )	Determines whether TypeLib information is present for the specified type of object
(vlax-write-enabled-p <i>obj</i> )	Determines whether an AutoCAD drawing object can be modified

# Reactor Functions

Reactor functions define, query, and delete reactors and reactor properties.

**NOTE** Before you can use these functions, you must load AutoLISP reactor support by issuing the following command:

```
(vl-load-com)
```

The `vl-load-com` function initializes reactor support and a number of other AutoLISP extensions.

Reactor functions	
Function	Description
(vl-load-com)	Loads AutoLISP reactor support functions and other AutoLISP extensions
(vlr-acdb-reactor <i>data callbacks</i> )	Constructs a database (global) reactor object
(vlr-add <i>obj</i> )	Enables a disabled reactor object
(vlr-added-p <i>obj</i> )	Tests to determine whether a reactor object is enabled
(vlr-beep-reaction [ <i>args</i> ])	Produces a beep sound
(vlr-current-reaction-name)	Returns the name (symbol) of the current event, if called from within a reactor's callback
(vlr-data <i>obj</i> )	Returns application-specific data associated with a reactor
(vlr-data-set <i>obj data</i> )	Overwrites application-specific data associated with a reactor
(vlr-deepclone-reactor <i>obj data</i> )	Constructs an editor reactor object that notifies of deep clone events
(vlr-docmanager-reactor <i>obj data</i> )	Constructs a reactor object that notifies of MDI-related events
(vlr-dwg-reactor <i>obj data</i> )	Constructs an editor reactor object that notifies of a drawing event (for example, opening or closing a drawing file)

## Reactor functions (*continued*)

Function	Description
(vlr-dxf-reactor <i>obj data</i> )	Constructs an editor reactor object that notifies of an event related to reading or writing of a DXF file
(vlr-editor-reactor <i>data callbacks</i> )	Constructs an editor (global) reactor object
(vlr-linker-reactor <i>data callbacks</i> )	Constructs a linker (global) reactor object
(vlr-miscellaneous-reactor <i>data callbacks</i> )	Constructs an editor reactor object that does not fall under any of the other editor reactor types
(vlr-mouse-reactor <i>data callbacks</i> )	Constructs an editor reactor object that notifies of a mouse event (for example, a double-click)
(vlr-notification <i>reactor</i> )	Determines whether or not a reactor's callback function will execute if its associated namespace is not active
(vlr-object-reactor <i>owners data callbacks</i> )	Constructs an object reactor object
(vlr-owner-add <i>reactor owner</i> )	Adds an object to the list of owners of an object reactor
(vlr-owner-remove <i>reactor owner</i> )	Removes an object from the list of owners of an object reactor
(vlr-owners <i>reactor</i> )	Returns the list of owners of an object reactor
(vlr-pers <i>reactor</i> )	Makes a reactor persistent
(vlr-pers-list [ <i>reactor</i> ])	Returns a list of persistent reactors in the current drawing
(vlr-pers-p <i>reactor</i> )	Determines whether or not a reactor is persistent
(vlr-pers-release <i>reactor</i> )	Makes a reactor transient
(vlr-reaction-names <i>reactor-type</i> )	Returns a list of all callback conditions for this reactor type
(vlr-reaction-set <i>reactor event function</i> )	Adds or replaces a callback function in a reactor
(vlr-reactions <i>reactor</i> )	Returns a list of pairs ( <i>event-name . callback_function</i> ) for the reactor
(vlr-reactors [ <i>reactor-type...</i> ])	Returns a list of reactors of the specified types

## Reactor functions (*continued*)

Function	Description
(vlr-remove <i>reactor</i> )	Disables a reactor
(vlr-remove-all <i>reactor-type</i> )	Disables all reactors of the specified type
(vlr-set-notification <i>reactor</i> <i>'range'</i> )	Defines whether or not a reactor's callback function will execute if its associated namespace is not active
(vlr-sysvar-reactor <i>data</i> <i>callbacks</i> )	Constructs an editor reactor object that notifies of a change to a system variable
(vlr-toolbar-reactor <i>data</i> <i>callbacks</i> )	Constructs an editor reactor object that notifies of a change to the bitmaps in a toolbar
(vlr-trace-reaction)	A pre-defined callback function that prints one or more callback arguments in the Trace window
(vlr-type <i>reactor</i> )	Returns a symbol representing the reactor type
(vlr-types)	Returns a list of all reactor types
(vlr-undo-reactor <i>data</i> <i>callbacks</i> )	Constructs an editor reactor object that notifies of an undo event
(vlr-wblock-reactor <i>data</i> <i>callbacks</i> )	Constructs an editor reactor object that notifies of an event related to writing a block
(vlr-window-reactor <i>data</i> <i>callbacks</i> )	Constructs an editor reactor object that notifies of an event related to moving or sizing an AutoCAD window
(vlr-xref-reactor <i>data</i> <i>callbacks</i> )	Constructs an editor reactor object that notifies of an event related to attaching or modifying XREF

# VLX Namespace Functions

The VLX namespace functions listed below apply to separate-namespace VLX applications. These functions allow separate-namespace VLX functions to be accessible from a document namespace, enable the retrieval and updating of variables in the associated document namespace, and provide error-handling routines for separate-namespace VLX functions.

VLX namespace functions	
Function	Description
(vl-arx-import [ <i>function</i>   <i>application</i> ])	Imports ADS-DEFUN functions into a separate-namespace VLX
(vl-doc-export ' <i>function</i> )	Makes a function loaded in a VLX namespace available to the current document
(vl-doc-import [' <i>function</i>   <i>application</i> ])	Imports a function that was previously exported from another separate-namespace VLX
(vl-doc-ref <i>symbol</i> )	Retrieves the value of a variable from the namespace of the associated document
(vl-doc-set <i>symbol value</i> )	Sets the value of a variable in the associated document's namespace
(vl-exit-with-error " <i>msg</i> ")	Passes control from a VLX error handler to the <b>*error*</b> function of the associated document namespace
(vl-exit-with-value <i>value</i> )	Returns a value to the document namespace from which the VLX was invoked
(vl-list-exported-functions [ <i>"appname"</i> ])	Lists all functions exported by the specified application, or all exported functions if no application is specified
(vl-list-loaded-vlx)	Returns a list of all separate-namespace VLX files associated with the current document
(vl-unload-vlx " <i>appname</i> ")	Unloads a VLX that is loaded in its own namespace (a separate-namespace VLX)
(vl-vlx-loaded-p " <i>appname</i> ")	Determines whether a VLX is loaded in its own namespace

# Namespace Communication Functions

The namespace communication functions consist of blackboard addressing and multi-document-loading functions.

## Namespace communication functions

Function	Description
(vl-bb-ref ' <i>variable</i> )	Returns the value of a variable from the blackboard namespace
(vl-bb-set ' <i>variable value</i> )	Sets the value of a variable in the blackboard namespace
(vl-load-all " <i>filename</i> ")	Loads a file into all open AutoCAD documents, and into any document subsequently opened during the current AutoCAD session
(vl-propagate ' <i>variable</i> )	Copies the value of a variable into all open AutoCAD documents, and into any document subsequently opened during the current AutoCAD session



# Windows Registry Functions

Windows Registry functions query and update the Windows Registry.

Windows Registry functions	
Function	Description
(vl-registry-delete <i>reg-key</i> [ <i>val-name</i> ])	Deletes the specified key or value from the Windows Registry
(vl-registry-descendents <i>reg-key</i> [ <i>val-names</i> ])	Returns a list of subkeys or value names for the specified Registry key
(vl-registry-read <i>reg-key</i> [ <i>val-name</i> ])	Returns data stored in the Windows Registry for the specified key/value pair
(vl-registry-write <i>reg-key</i> [ <i>val-name val-data</i> ])	Creates a key in the Windows Registry
(vlax-product-key)	Returns the AutoCAD registry path <b>NOTE</b> This is an extended function provided by VLISP. You must issue <b>vl-load-com</b> before you can use the function.



# Visual LISP Environment and Formatting Options

This chapter describes the configuration options available on the VLISP Tools menu. The Window Attributes menu item sets color-coding options for the VLISP text editor windows. The Environment Options item allows you to set session-wide VLISP options (for example, whether or not to create automatic backup files, or how to treat protected symbols), diagnostic options (such as what statistics to report during syntax checking, or what level of detail to display when inspecting drawing objects), formatting options for AutoLISP code, and page layout options for printed output.

## In This Appendix

# B

- Window Attributes
- Environment Options
- Save Settings

# Window Attributes

The Window Attributes submenu includes selections for customizing the VLISP windowing environment, controlling attributes such as colors, fonts, and code formatting. The Syntax Coloring, Current to Prototype, and All to Prototype items are available only for text editor windows.

VLISP allows you to define prototype configurations for text editor windows. The prototype becomes the default configuration for these windows. For example, when you open a new file in the VLISP text editor, the editor window assumes the attributes and properties of the prototype editor configuration. The window prototype includes

- Color scheme
- Lexical coloring flag
- Tab size
- Left margin indent

Every time you change and save any text editor window attribute settings, VLISP will ask you if the modified setting should be used as a prototype for this window type.

## Syntax Coloring

The Syntax Coloring item determines the type of syntax coloring that will be used for the current file being edited. This option is available when you edit a file whose file type is not *.lsp*. When chosen, Syntax Coloring displays the Color Style dialog box, which provides the following options:

None	No color coding.
AutoLISP	Use LISP syntax color coding. This color coding scheme is used for all files of type <i>.lsp</i> .
C++	Use C++ syntax color coding. This is the default for all files of type <i>.cpp</i> , <i>.c++</i> , <i>.c</i> , <i>.hpp</i> , <i>.h++</i> , and <i>.h</i> .
DCL	Use DCL syntax color coding. This is the default for all files of type <i>.dcl</i> .
SQL	Use SQL syntax color coding. This is the default for all files of type <i>.sql</i> .

If a file type does not have a default coloring style, the user is asked whether to use the selected coloring style for all files of the same type.

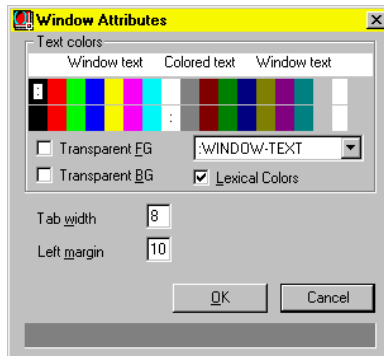
---

**NOTE** All formatting and “smart” indent features require the AutoLISP lexical coloring style.

---

## Configure Current

This command allows you to configure the attributes of the current window. It is applicable to the VLISP text editor and Console windows. The Configure Current command displays the Window Attributes dialog box:



This dialog box lets you customize the tab width and left margin sizes, customize various text colors defined for the current window type, and control the lexical coloring for that window (if applicable). To select the color with the aid of the color selection control, click the mouse button in the rectangle that is painted with the color you want to set.

**Text Colors**      The upper row of rectangles indicates foreground color, the lower row indicates background color. When you select a color, the color palette changes its color with respect to your choice. Use the pull-down list to select the attribute of the window whose colors you want to change.

The available choices depend on which window is current. The following are possible window attributes:

- *:Input-Zone*. The input area following the Console window prompt.
- *:Window-Text*. Text displayed in the window (other than input zone).
- *:Window-Selection*. Selected text.
- *:Error-Highlight*. Error messages in the Build Output window.
- *:Console-Message*. No effect (reserved for future use).
- *:BPT-Active*. Active breakpoint.
- *:BPT-Disable*. Disabled breakpoint.

The pull-down list may also provide options for changing the lexical coloring of the following AutoLISP code components:

- *:LEX-SPACE*. Spaces.
- *:LEX-STR*. Strings.
- *:LEX-SYM*. Symbols.
- *:LEX-NUM*. Reserved for future use.
- *:LEX-INT*. Integers.
- *:LEX-REAL*. Real numbers.
- *:LEX-COMM*. Reserved for future use.
- *:LEX-COMM1*. Comments that begin with one or more semicolons.
- *:LEX-COMM2*. Inline and multi-line comments (comments that begin with ; | and end with | ;).
- *:LEX-PAREN*. Parentheses.
- *:LEX-SPEC*. Reserved for future use.
- *:LEX-SPEC1*. Reserved for future use.
- *:LEX-UNKN*. Unknown items.

Transparent FG	Transparent foreground.
Transparent BG	Transparent background.
Lexical Colors	If selected, VLISP applies the selected color coding options. If you want to use the VLISP formatter but do not want lexical coloring, turn this option off.
Tab Width	Sets tab spacing in the current window.
Left Margin	Sets the left margin of the current window.

When you change and save the configuration of a VLISP editor window, you will be asked whether or not you want the configuration to become the prototype for all text editor windows. If you choose Yes, the configuration of the current window becomes the new prototype for VLISP text editor windows, and all open text editor windows assume the attributes of the prototype.

## Set Current Window to Prototype

This command configures the current active window with the attributes of the prototype window.

## Set All Windows to Prototype

This command sets all open windows with the attributes of the prototype.

## Font

This option opens a standard Windows Font dialog box for selecting the font to be used in VLISP windows.

Note that for code formatting to work correctly, you must use a fixed (non-proportional) font.

# Environment Options

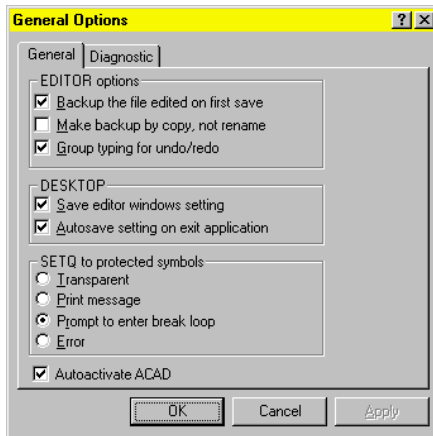
The Environment Options menu item allows you to set session-wide VLISP options. For example, you can tell VLISP whether to save text editor files at set intervals automatically, whether to create automatic backup files, and how you want to treat attempts to modify protected symbols. Environment Options is also where you set diagnostic options, such as what statistics to report during syntax checking, and what level of detail to display when inspecting drawing objects. You can also set formatting options for AutoLISP code, and page layout options for printed output.

## General Options

The General Options menu item displays a tabbed dialog box containing General and Diagnostic tabs.

### General Options Tab

The General Options tab contains three groups of items: Editor Options, Desktop Options, and SETQ to Protected Symbols.



The Editor Options group contains the following items:

Backup the File Edited on First Save	When this option is selected, VLISP creates a backup copy of the file the first time you save it.
Make Backup by Copy, Not Rename	VLISP creates a copy of the original file. When this option is not selected, the backup file is a renamed version of the original file.
Group Typing for Undo/Redo	VLISP groups keystrokes for the Undo and Redo Edit commands. If not selected, Undo and Redo proceed one character at a time.

The Desktop Options group contains the following items:

Save Editor Windows Settings	The VLISP text editor window settings (toolbar placement and text/background color) will be saved along with window sizes, placements, and editor configurations upon issuing Save Settings. When the option is off, VLISP editor window settings are not saved on Save Settings.
Autosave Setting on Exit Application	Options are saved automatically upon exiting VLISP. If Save Editor Windows Settings is turned on, the VLISP text editor window settings (toolbar placement and text/background color) are saved along with window sizes, placements, and editor configuration.

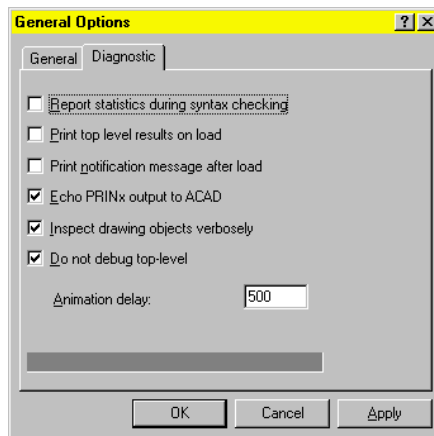


The SETQ to Protected Symbols group controls how VLISP responds to attempts to redefine protected symbols. (See “Protected Symbols” on page 228.) The options are:

Transparent	Protected symbols are treated like any other symbol.
Print Message	AutoLISP issues a warning message when you modify a protected symbol but carries out the modification.
Prompt to Enter Break Loop	This is the default option. AutoLISP displays a message box asking whether or not to enter a break loop when you attempt to modify a protected symbol. If you choose No, the symbol's value is modified and processing continues normally. If you choose Yes, processing is interrupted and you enter a VLISP break loop.  In a break loop, control switches to the VLISP Console window. You can set the symbol and continue processing by pressing the Continue button on the VLISP toolbar, or abort modification by pressing Reset.
Error	Prohibits modification of protected symbols. Any attempt to modify a protected symbol results in an error.

## Diagnostic Options

The Diagnostic Options tab displays the following dialog box:

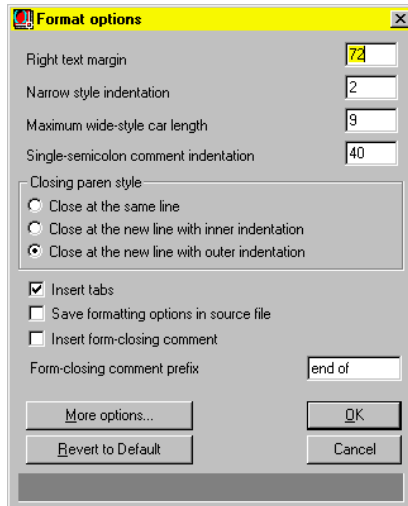


Diagnostic options are as follows:

Report Statistics During Syntax Checking	If selected, the syntax checker and the file compiler report the statistics after checking or compiling each top-level form and after each file.
Print Top Level Results On Load	<p>If selected, top-level expressions are evaluated and printed in the Console window when the expressions are loaded.</p> <p>A top-level expression is one that appears outside any other expression (for example, an expression that appears outside of <code>defun</code>). For example, the following call to <code>list</code> is a top-level expression:</p> <pre>(list 1 2 3) (defun foo (x) x)</pre> <p>Loading a file containing this code results in the following being printed in the Console window, if Print Top Level Results on Load is selected:</p> <pre>(1 2 3) FOO</pre>
Print Notification Message After Load	If selected, each call to the <code>load</code> function results in messages printed to the VLISP Console window.
Echo PRINx Output to ACAD	If selected, the default output of functions <code>print</code> , <code>prin</code> and <code>prin1</code> is echoed to the AutoCAD Command window and the VLISP Console window instead of only to the VLISP Console window.
Inspect Drawing Objects Verbosely	If selected, the elements of an object are included in the Inspect window for a drawing database object. If the option is not selected, only the object line (containing the entity name) appears in drawing object Inspect windows.
Do Not Debug Top-Level	If selected and Stop Once mode is set, the break will not occur before evaluating every top-level form (such as <code>defun</code> ) during the file load process.
Animation Delay	Determines the pause length between program steps in Animate mode, measured in milliseconds. The default is 100.

# Visual LISP Format Options

The Visual LISP Format Options menu item displays the Format Options dialog box used to set formatting and indentation options.



Additional formatting options are available by pressing the More Options button. These options are covered in “Additional Formatting Options” on page 481.

Pressing Revert to Default reverts the option settings to the previous saved setting, or to the system default, if the user has not previously changed and saved settings.

Note that the code formatter assumes that you are using a fixed font to display or print formatted text.

## Base Formatting Options

Right Text Margin	Right margin for text. An expression prints on a single line if its last character position does not exceed the right margin.
Narrow Style Indentation	Sets the standard indentation value used in the Narrow Formatting Style for function arguments.

Maximum Wide-Style Car Length	Maximum length for function expressions in Wide Formatting Style. For longer expressions, the formatter always uses Narrow Formatting Style.
Single-Semicolon Comment Indentation	Identifies the left margin alignment for single-semicolon comments.

The Closing Parenthesis Style group of options controls the position of closing parentheses for multi-line formatting styles. The effect of each option is demonstrated by formatting the following code, where Right Text Margin is set to 40 and Preserve Existing Line Breaks is not selected:

```
(cond
  ((/= (logand mask fl g) 0)
   (list (list txton)))
)
```

VLISP formats this code as follows:

Close at the Same Line	(cond ((/= (logand mask fl g) 0) (list (list txton))))
------------------------	---

Close at the New Line with Inner Indentation	(cond ((/= (logand mask fl g) 0) (list (list txton)) ) )
--	---

Close at The New Line with Outer Indentation	(cond ((/= (logand mask fl g) 0) (list (list txton)) ) )
--	---

The remaining items in this dialog box concern tabs, saving the current setting, and the setting of several comment options.

Insert Tabs	The VLISP formatter inserts tab characters instead of multiple space characters whenever possible.
Save Formatting Options in Source File	The VLISP formatter appends comments containing the current formatting settings to the end of the text in the VLISP text editor window. If you save these comments (and do not modify them), VLISP applies the saved formatting options to the text editor the next time you open the file.

Insert Form-Closing Comment	<p>This option causes VLISP to add a closing comment to an expression when Close at the New Line with Inner Indentation or Close at the New Line with Outer Indentation is selected. The comment takes the following form at the end of a multi-line function:</p> <pre>;_ end of &lt;function name&gt;</pre> <p>VLISP does not add the comment if the line already contains a comment.</p>
Form-Closing Comment Prefix	Determines the text to be included when Insert Form-closing Comment is selected.

## Additional Formatting Options

These options appear when you press the More Options button on the basic formatting window.

Approximate Line Length	The VLISP formatter checks this value when it chooses the formatting style. If the number of characters in an expression is greater than this value, the formatter prints the expression as multi-line.
Preserve Existing Line Breaks	<p>When this option is selected, the VLISP formatter inserts new lines whenever a new line is detected in the text being formatted. When the option is off, the formatter can squeeze a multiple-line expression to the plane style, if it fits within the right margin.</p> <p><b>NOTE</b> Selecting this option causes VLISP to ignore other formatting options when those options would result in a loss of existing line breaks. This is often the source of unexpected formatting results.</p>
Split Comments	When this option is selected, the VLISP formatter splits comments that exceed the right margin.

Casing for  
Symbols

These options control whether or not the VLISP formatter converts the case of alphabetic text in an AutoLISP symbol name. The protected symbols subgroup controls the case conversion of protected symbols (that is, symbols with the ASSIGN-PROTECT flag set). The unprotected options subgroup controls the case conversion of unprotected AutoLISP symbols. The following options are available:

*None*: No case conversion.

*downcase*: The formatter converts all characters in a symbol name to lowercase.

*UPCASE*: The formatter converts all characters in a symbol name to uppercase.

The Long List format style option controls the formatting of long lists. Long lists are lists of formal arguments in `defun` and `lambda` expressions, or in quoted lists of atoms with more than five elements. The style option applies to long lists that do not fit on a single line (that is, within the Right Text Margin). Long List format style options are illustrated by formatting the following statement with Right Text Margin set to 45:

```
(setq lista '("abc" "def" "ghi" "jkl" "mno" "pqr"))
```

The options are:

Single-  
Column  
Formatting

```
(setq lista '("abc"
              "def"
              "ghi"
              "jkl"
              "mno"
              "pqr"
              )
)
```

2-Column  
Formatting

```
(setq listall '("abc" "def"
                "ghi" "jkl"
                "mno" "pqr"
                )
)
```

Multi-Column  
Formatting

```
(setq listall '("abc" "def" "ghi"
                "jkl" "mno" "pqr"
                )
)
```

Fill-to-Margin  
Formatting

```
(setq listall '("abc" "def" "ghi" "jkl" "mno"
                "pqr"
                )
)
```

## Page Format Options

These options control the appearance of printed output from VLISP. The options are:

Header and Footer	Fields for specifying the content of the page header and footer. The first input field contains the header line to print at the top of the page, the second input field contains the footer line to print at the bottom of the page. Headers and footers may contain text strings and variables. See “Justification and Substitution Codes” on page 483 for more information on specifying headers and footers.
Print Page Margins	Select either inches or millimeters as the measuring unit, then specify Top, Left, Bottom, and Right margins.

Press the Font button to open a dialog box for specifying the output font.

### Justification and Substitution Codes

Only the ampersand character (&) is considered as an escape character. Both the ampersand and the character that follows are not copied to the output line. Instead, they either justify the following text, or VLISP replaces them with variable information.

Justification codes indicate how header text is justified on the printed page. You can specify any of the following:

&l	Left-justified (the default).
&c	Centered within the page margins.
&r	Right-justified.

Replace codes are variables for which VLISP substitutes values. Specify any of the following:

&f	VLISP replaces this code with the title of the active VLISP window. When invoked from a VLISP text editor window, <i>&amp;f</i> is the name of the file being edited, including directory and extension.
&d	Current system date. To select a date format, choose Edit ► Extra Commands ► Format Date/Time from the VLISP menu.
&t	Current system time. To select a time format, choose Edit ► Extra Commands ► Format Date/Time from the VLISP menu.
&p	Current page number.

To include an ampersand character in your heading text, enter two in succession.

The default heading is set to the following:

&cFile: &f &r&dt

The default footing is set as follows:

&r&p

The following is a sample printed page from an AutoLISP source file using the default page layout settings:

```
File: REACTORSTUFF.LSP 12/11/98

(defun saveDrawingInfo (calling-reactor commandInfo / dwgname
fileSize)
  (setq dwgname (cadr commandInfo)
        fileSize (vl-file-size dwgname)
  )
  (alert (strcat "The file size of " dwgname " is "))
  .
  .
  .
  (princ)
)
```

2

## Save Settings

The Save Settings tools option saves the desktop configuration and options settings. Note that the desktop configuration concerning the child windows attributes (their presence on the screen, color, position, files loaded) is saved only when the Save Editor Windows Settings option is on. This option is available through the Desktop Options tab of the General Options dialog box.



# AutoLISP Error Codes

This appendix lists the AutoLISP error codes.



In This Appendix

C



■ Error Codes

# Error Codes

The following table shows the values of error codes generated by AutoLISP. The `ERRNO` system variable is set to one of these values when an AutoLISP function call causes an error that AutoCAD detects. AutoLISP applications can inspect the current value of `ERRNO` with `(getvar "errno")`.

The `ERRNO` system variable is not always cleared to zero. Unless it is inspected immediately after an AutoLISP function has reported an error, the error that its value indicates may be misleading. This variable is always cleared when starting or opening a drawing.

**NOTE** The possible values of `ERRNO`, and their meanings, may change in future releases of AutoCAD.

Online program error codes	
Value	Meaning
0	No error
1	Invalid symbol table name
2	Invalid entity or selection set name
3	Exceeded maximum number of selection sets
4	Invalid selection set
5	Improper use of block definition
6	Improper use of xref
7	Object selection: pick failed
8	End of entity file
9	End of block definition file
10	Failed to find last entity
11	Illegal attempt to delete viewport object
12	Operation not allowed during PLINE

**Online program error codes (continued)**

Value	Meaning
13	Invalid handle
14	Handles not enabled
15	Invalid arguments in coordinate transform request
16	Invalid space in coordinate transform request
17	Invalid use of deleted entity
18	Invalid table name
19	Invalid table function argument
20	Attempt to set a read-only variable
21	Zero value not allowed
22	Value out of range
23	Complex REGEN in progress
24	Attempt to change entity type
25	Bad layer name
26	Bad linetype name
27	Bad color name
28	Bad text style name
29	Bad shape name
30	Bad field for entity type
31	Attempt to modify deleted entity
32	Attempt to modify seqend subentity
33	Attempt to change handle
34	Attempt to modify viewport visibility
35	Entity on locked layer

**Online program error codes (continued)**

Value	Meaning
36	Bad entity type
37	Bad polyline entity
38	Incomplete complex entity in block
39	Invalid block name field
40	Duplicate block flag fields
41	Duplicate block name fields
42	Bad normal vector
43	Missing block name
44	Missing block flags
45	Invalid anonymous block
46	Invalid block definition
47	Mandatory field missing
48	Unrecognized extended data (XDATA) type
49	Improper nesting of list in XDATA
50	Improper location of APPID field
51	Exceeded maximum XDATA size
52	Entity selection: null response
53	Duplicate APPID
54	Attempt to make or modify viewport entity
55	Attempt to make or modify an xref, xdef, or xdep
56	ssget filter: unexpected end of list
57	ssget filter: missing test operand
58	ssget filter: invalid opcode (-4) string

**Online program error codes (continued)**

Value	Meaning
59	ssget filter: improper nesting or empty conditional clause
60	ssget filter: mismatched begin and end of conditional clause
61	ssget filter: wrong number of arguments in conditional clause (for NOT or XOR)
62	ssget filter: exceeded maximum nesting limit
63	ssget filter: invalid group code
64	ssget filter: invalid string test
65	ssget filter: invalid vector test
66	ssget filter: invalid real test
67	ssget filter: invalid integer test
68	Digitizer is not a tablet
69	Tablet is not calibrated
70	Invalid tablet arguments
71	ADS error: Unable to allocate new result buffer
72	ADS error: Null pointer detected
73	Cannot open executable file
74	Application is already loaded
75	Maximum number of applications already loaded
76	Unable to execute application
77	Incompatible version number
78	Unable to unload nested application
79	Application refused to unload
80	Application is not currently loaded
81	Not enough memory to load application

**Online program error codes (*continued*)**

Value	Meaning
82	ADS error: Invalid transformation matrix
83	ADS error: Invalid symbol name
84	ADS error: Invalid symbol value
85	AutoLISP/ADS operation prohibited while a dialog box was displayed

# Index

## Symbols

- ! (exclamation point), for viewing AutoCAD variable values, 18, 25
- \* (asterisk)
  - as Apropos wild-card character, 44
  - by file name in text editor status bar, 14
- \*obj\* system variable
  - copying Inspect objects to, 103
  - copying symbol name from Apropos, 47
  - copying trace stack elements to, 88
  - copying value from Frame Binding window to, 88
  - copying value from Inspect window, 102
  - copying value from Watch window, 84
- ; (semicolon) in VLISP comment styles, 58
- \_ (underscore)
  - for foreign-language support, 261
  - for VLISP comments, 58
- \ (backslash)
  - for control characters in quoted strings, 237–238
  - using within quoted strings, 237
- \*error\* function
  - for document namespaces, 134, 135
  - overview, 254–257
- " (quotation marks)
  - and parentheses in expressions, 223
  - using within quoted strings, 237
- ' (single quotation mark), forming point lists, 243
- // (slashes) for DCL comments, 344
- .\_xx files, 139
- | (vertical bar) for VLISP comments, 58

## A

- acad.dcl* file, 340
- acad.unt*. *See* unit definition file
- :ACAD-REQUEST trace keyword frame, 89
- accessing
  - AutoCAD groups, 334
  - AutoCAD objects with ActiveX, 165–169
    - circle object example, 168–169

## accessing (*continued*)

- moving forward from the application
  - object, 168
- overview, 165, 168–169
- returning pointer to AutoCAD object, 165
  - specifying VLA-object type, 165
  - using Inspect tool to view object properties, 166–169
- dictionary entries, 334
- entities, 311
- help files, 290–291
- symbol table entries, 332–333
- user input, 291
  - calibrating tablets, 291–293
  - summarized, 447
- :ACMD-CALLBACK trace keyword frame, 90
- action (DCL attribute), 397
- action expressions (dialog boxes). *See* dialog box management, action expressions and callbacks
- action\_tile function, 372, 374
- activating. *See* starting
- active document, changing, 28
- ActiveX, 161–199
  - accessing AutoCAD objects, 165–169
    - circle object example, 168–169
    - moving forward from the application object, 168
  - overview, 165, 168–169
  - returning pointer to AutoCAD object, 165
    - specifying VLA-object type, 165
    - using Inspect tool to view object properties, 166–169
- AutoCAD object model, 162–164
  - collections of objects, 164
  - object methods, 164
  - object properties, 164
- AutoLISP functions and, 162
- connecting to application objects, 165
- connecting to applications, 195

## ActiveX (*continued*)

- data types
  - converting from AutoLISP to ActiveX, 173–180
- enabling AutoLISP support for, 22
- errors, handling of using AutoLISP, 191–193
- interacting with other applications, 193–199
  - calling ActiveX methods with `vlax-invoke-method`, 198–199
  - establishing connection to application, 195
  - importing a type library, 193–195
  - obtaining properties with `vlax-get-property`, 199
  - overview, 193
  - sample application, 196–198
  - updating properties with `vlax-put-property`, 199
  - without importing a type library, 198–199
- intercepting errors returned from, 191–193
- languages and environments supporting, 162
- loading ActiveX support for AutoLISP, 22, 165
- methods, 164–173, 198–199
  - in ActiveX and VBA Reference, 164, 171–174, 193
- objects. *See* objects
- overview, 162
- properties of objects
  - in ActiveX and VBA Reference, 164
  - changing, 181–183
  - getting, 180–181
  - listing, 185–187
  - reading, 180–181
  - updating, 181–183
  - viewing, 180–181
- safearrays, using
  - creating, 175–177
  - creating variants containing, 178–179
  - displaying contents of, 177
  - populating with data, 175–178
  - with variants, 178–179
- using AutoLISP functions with ActiveX
  - methods, 170–191
  - applying object's property to new object, 181
  - AutoLISP data types accepted in place of ActiveX data type, 180
  - changing X axis of circle, 181–183
  - converting AutoLISP data types to ActiveX data types, 173–180
  - converting object references, 190–191

## ActiveX (*continued*)

- determining function needed, 172
- determining how to call functions, 172–173
- determining if method or property applies to object, 186–187
- determining if object is available for updating, 183
- listing object properties and methods, 185–187
- reading object properties, 180–181
- releasing objects and freeing memory, 189
- updating object properties, 181–183
- using ActiveX methods that return values in arguments, 184–185
- `vla-` functions, 170
- `vla-get` functions, 170
- `vla-put` functions, 170
- `vlax-` functions, 170
- working with collection objects, 187–189
- variants, using, 174–175
  - changing data type of, 174
  - creating, 174–175
  - returning data type of, 174
  - returning value of, 174
  - safearrays and, 178–179
- ActiveX and VBA Developer's Guide, 193
- ActiveX and VBA Reference, 8, 164, 171–174, 193
- Add File command (Project window shortcut menu), 146
- Add Lisp Source Files to Project dialog box, 120
- Add to Watch command
  - Apropos results window shortcut menu, 47
  - Frame Binding window shortcut menu, 89
- Add Watch command
  - Console window shortcut menu, 29
  - text editor shortcut menu, 33
- `add_list` function, 380–381
- `adsfunc` function, 305
- aesthetics in dialog box design, 353
- `:AFTER-EXP` trace keyword frame, 90
- `alert` function, 256, 355, 364
- alignment (DCL attribute), 397–426
- All (Apropos filter value), 44
- `allow_accept` (DCL attribute), 398
- angles
  - converting radians and degrees, 281–282
  - converting to strings, 280
  - finding angle between line and X axis, 274
- `angtof` function, 281
- `angtos` function, 280, 281
- angular values, converting radians or degrees, 281–282



- Animate command (VLISP Debug menu), 74
- Animate mode, 74
- anonymous blocks, 319
- append function, 241
- Append With text formatting option, 59
- appending information to existing Console log file, 30
- Application Compilation Options dialog box, 122–123
- Application Directory dialog box, 118
- application modules, making. *See* making application modules
- Application Option dialog box, 118
- Application Properties dialog box, 124
- Application Wizard. *See* Make Application Wizard
- application-handling functions, 436
- applications
  - building. *See* building applications
  - managing. *See* managing VLISP applications
  - running, 114–115, 124
  - separate-namespaces, 129–136
- Apropos dialog box, 43–44
- Apropos dialog box options, 43–44
- Apropos feature, 43–47
  - Apropos dialog box, 43–44
  - calling from Watch window, 85
  - Complete Word by Apropos feature, 48–49
  - Copy to Trace/Log feature, 46
  - Downcase Symbols option, 44
  - Filter Flags option, 44
  - Filter Values option, 44
  - inserting symbols from results window, 46
  - invoking from Console window shortcut menu, 29
  - invoking from text editor shortcut menu, 33
  - invoking from Watch window shortcut menu, 85
  - Match by Prefix option, 45
  - opening Apropos window, 29, 33, 43
  - overview, 43
  - results window, 45–47
  - searching for AutoLISP symbols by prefix, 45
  - shortcut menu options, 46–47
  - Use WCMATCH option, 44
  - using search results, 45–47
- Apropos Window command (VLISP View menu), 43
- arbitrary data management, 331
- arithmetic functions, 437–438
- :ARQ-SUBR-CALLBACK trace keyword frame, 90
- Arrange Files command (Project window shortcut menu), 146
- arranging files in Project window, 146
- arrays. *See* safearrays
- ARX applications. *See* ObjectARX applications
- ARX Object ID, 190, 191
- ASCII code conversions, 282–283
- aspect\_ratio (DCL attribute), 398
- assoc function, 244, 245
- association lists, 244
- associative searches. *See* searching
- asterisk (\*)
  - as Apropos wild-card character, 44
  - by file name in text editor status bar, 14
- atoi function, 382
- attaching data
  - extended data to entities, 329
  - to reactor objects, 210
- attaching reactors to AutoCAD drawings. *See* reactors
- attributes, DCL. *See* DCL
- AutoCAD
  - accessing AutoCAD groups, 334
  - active document, 28
  - blocks, viewing with VLISP, 106–107
  - calibrating tablets, 291–293
  - commands
    - issuing with AutoLISP, 260–261
    - redefining, 249–250
  - configuration control, 264
  - coordinate systems, 286–287
  - device access and control functions, 291–293
  - display control, 264–268
    - graphics and text windows, 267
    - low-level graphics, 267–268
    - overview, 264
  - entities, viewing with VLISP, 104–105
  - exiting, 22
  - foreign-language support, 261
  - geometric utilities, 274–279
    - finding angle between line and X axis, 274
    - finding distance between two points, 274
    - finding intersection of two lines, 274
    - finding polar coordinates of points, 274
    - overview, 274
  - getting user input from, 291
  - handling user input, 268–273
    - pausing for input, 261–262
  - help files, accessing, 290–291
  - inspecting and changing system and environment variables, 263–264
  - menus, controlling, 264–267
  - object snap, 274–275
  - objects, manipulating, 295–334
    - extended data, 323–330
    - object-handling, 305–323
    - selection set handling, 296–305

## AutoCAD (*continued*)

- symbol table and dictionary access, 332–334
  - xrecord objects, 331
  - passing pick points to commands, 262–263
  - pausing for user input, 261–262
  - query and command functions, 260–264
  - receiving user input from, 291
  - redefining AutoCAD commands, 249–250
  - related publications, 8
  - running with VLISP, 3, 20
  - sending commands to AutoCAD prompt, 260–261
  - text extents, 275–279
  - undoing commands issued with command function, 263
  - user input functions, 268–273
    - accessing user input from devices, 291
    - allowable input, 268–269
    - arbitrary keyboard input, 273
    - controlling user-input function conditions, 271–273
    - getting user input, 268–271
    - getxxx functions, 268–271
    - input options, 271–272
    - keyword options, 272–273
    - pausing for user input, 261–262
    - validating input, 273
  - viewing drawing entities, 104–108
    - blocks in the drawing database, 106–107
    - controlling amount of Inspect information, 104
    - entities in the drawing database, 104–105
    - extended data associated with objects, 105, 107–108
    - selected objects in drawings, 107
    - symbol tables in the drawing database, 105
- ## AutoCAD command window
- Console window compared to, 18, 25–26, 28
  - starting Visual LISP, 12
- ## AutoCAD drawing database, viewing
- blocks in, 106–107
  - entities in, 104–105
  - extended data, 108
  - selected objects in drawings, 107
  - symbol tables in, 105
- ## AutoCAD groups, accessing, 334
- ## AutoCAD mode command (Console window shortcut menu), 29
- ## AutoCAD object model, 162–164
- collections of objects, 164
  - object methods, 164
  - object properties, 164

## AutoCAD objects. *See* objects

## AutoCAD Runtime Extension. *See* ARX

## Autodesk World Wide Web site, 8

## AutoLISP

- accessing AutoCAD groups, 334
- accessing user input with, 291
- application-handling functions, 436
- arithmetic functions, 437–438
- AutoCAD display control, 264–268
  - graphics and text windows, 267
  - low-level graphics, 267–268
  - menus, 264–267
  - overview, 264
- calibrating tablets with, 291–293
- closing files in programs, 227
- collection manipulation functions, 458
- comments in program files, 230
- communicating with AutoCAD, 259–293
  - converting data types and units, 279–289
  - device access and control, 291–293
  - display control, 264–268
  - file-handling functions, 289–291
  - geometric utilities, 274–279
  - getting user input, 268–273
  - query and command functions, 260–264
- compiling and linking AutoLISP programs, 110–116
  - Application Compilation Options dialog box, 122–123
  - choosing a compilation mode, 111, 154, 155
  - choosing build options, 144, 153–155
  - compile example, 113–114
  - compiling programs from files, 111–113
  - defining build options, 153–155
  - linking function calls, 111, 116, 159
  - loading and running compiled programs, 114–115
  - localizing variables, 155
  - Make Application output, 123
  - Merge Files mode, 154
  - naming an output file, 113
  - optimization during compiling, 111
  - project source files, 146–147
  - recompiling project source files, 148
  - report detail level, 155
  - safe optimization, 155, 157–160
  - using Application Wizard, 117–123
  - using the compiler, 110
  - using VLISP integrated project management facilities, 110
- conditional branching and looping, 240
- control characters in strings, 237–238
- controlling dialog boxes with, 366–368

## AutoLISP (*continued*)

- converting data types and units, 279–289
  - angular values from radians or degrees, 281–282
- ASCII code conversions, 282–283
- coordinate system transformations, 286–289
- measurement unit conversions, 283–286
- point transformations, 289
- string conversions, 279–281
- synopsis of functions, 446, 458–459
- curve measurement functions, 461–462
- data types, 224–229
  - entity names, 226
  - file descriptors, 226–227
  - integers, 224–225
  - lists, 225
  - reals, 225
  - selection sets, 226
  - strings, 225
  - symbols and variables, 227–229
  - VLA-objects, 226
- debugging AutoLISP code
  - break loop process, 76
  - Break on Error option, 75
  - breakpoint life cycle, 81
  - checking for syntax errors, 61–65
  - continuable break loops, 77–78
  - continuing program execution, 74, 77
  - Debug-on-Entry flag, 75, 94
  - disabling breakpoints temporarily, 80
  - displaying breakpoints, 80
  - example, 69–75
  - highlight colors for breakpoints, 79
  - listing breakpoints, 80
  - monitoring evaluation results of an expression, 73–74
  - non-continuable break loops, 78
  - optimization and bug introduction, 156
  - Quit Current Level option, 77
  - removing breakpoints, 79
  - Reset to Top Level option, 77
  - running in Animate mode, 74
  - semantic auditing of DCL files, 346–347, 387–388
  - setting breakpoints, 70–71, 79
  - starting debugging sessions, 76
  - starting Trace logging, 82
  - stepping through programs from breakpoints, 71–72, 78
  - Stop Once mode, 75, 76
  - Top-Level debugging mode, 75
  - using Frame Binding window, 88–92
  - using Inspect windows, 95–108

## AutoLISP (*continued*)

- using Symbol Service dialog box, 92–95
- using Trace Stack window, 85–92
- using Watch window, 83–85
- VLISP debugging features, 68–69
- developing AutoLISP programs, 24–65
  - basic steps, 24
  - checking for syntax errors, 61–65
  - formatting code, 50–60
  - using coding aids, 42–50
  - using the Console window, 25–30
  - using the text editor, 31–41
- device access and control functions, 291–293
  - synopsis of, 447
- dialog box management, 365–389
  - action expressions and callbacks, 368–372
  - application-specific data, 386
  - controlling dialog boxes with AutoLISP programs, 366–368
  - DCL error handling, 386–388
  - function sequence, 388–389
  - handling tiles, 372–376
  - hiding dialog boxes, 377–379
  - image tiles and buttons, 383–385
  - list boxes and pop-up lists, 380–382
  - nesting dialog boxes, 376–377
  - sample block definition dialog box, 389
- dictionary functions, 334, 462
- display-control functions, 448
- displaying messages with, 236
- dotted pairs, 244–245
- equality and conditional functions, 439
- equality verification, 240
- error codes, 485, 486
- error handling, 254–257, 440
- exiting quietly, 236
- expressions, 222–223
- extended data functions
  - attaching extended data to entities, 329
  - filtering selection sets for extended data, 300
  - group codes for extended data, 323–324
  - handles in extended data, 330
  - managing memory use, 329
  - organization of extended data, 324–326
  - registration of applications, 327
  - retrieving extended data, 323, 327–328
  - synopsis of, 453

## AutoLISP (*continued*)

- file-handling functions
  - accessing help files, 290–291
  - file search, 289–290
  - synopsis of, 449
- foreign-language support, 261
- formatting code, 229–230
- function synopsis (summary), 433–461
  - basic functions, 436–445
  - category summary, 434–436
  - memory management functions, 457
  - namespace communication functions, 468
  - reactor functions, 464–466
  - selection set, object, and symbol table functions, 453–457
  - utility functions, 446–452
  - Visual LISP extended functions, 457
  - VLX namespace functions, 467
  - Windows Registry functions, 469
- function syntax, 223
- function handling, 245–254
  - adding commands, 248–249
  - c:xxx functions, 247–250
  - defining functions, 245–247
  - defining functions with arguments, 252–254
  - defun function, 245–253
  - defun-q function, 247
  - local variables in functions, 251–252
  - redefining AutoCAD commands, 249–250
  - special forms, 253–254
  - synopsis of functions, 441
- functions, as lists, 246
- geometric utilities, 274–279
  - finding angle between line and X axis, 274
  - finding distance between two points, 274
  - finding intersection of two lines, 274
  - finding polar coordinates of points, 274
  - object snap, 274–275
  - overview, 274
  - synopsis of, 450
  - text extents, 275–279
- history of, 2
- integer overflow handling, 224–225
- interacting with users, 291
- issuing commands from Console window, 14
- list processing functions, 240–245
  - adding items to list beginning, 241, 241
  - combining lists, 241
  - dotted pairs, 244–245

## AutoLISP (*continued*)

- grouping related items, 240
- point lists, 242–244
- retrieving items from lists, 240–241
- returning all but first element, 241
- substituting items, 241, 241
- synopsis of, 442–443
- loading from text editor, 20
- manipulating AutoCAD objects, 295–334
  - extended data, 323–330
  - object handling, 305–323
  - selection set handling, 296–305
  - symbol table and dictionary access, 332–334
  - xrecord objects, 331
- matching parentheses, 222–223
- MDI (multiple document interface), and AutoLISP, 126–134
  - error-handling in MDI environment, 134–135
  - limitations, 136
  - namespaces overview, 126–129
  - running an application in its own namespace, 128–130
  - sharing data between namespaces, 133–134
- memory management functions, 457
- method invocation functions, 459
- namespace communication functions, 468
- number handling, 233
- object-handling functions, 305–323
  - blocks and, 308–311
  - entity access functions, 311
  - entity data functions, 311–320
  - entity name functions, 305–311
  - extended data, 323–330
  - non-graphic object-handling, 321–323
  - polylines (old-style and lightweight), 320–321
  - selection sets, 296–305
  - symbol table and dictionary entries, 332–334
  - synopsis of, 453–454, 463
  - xrecord objects, 331
- output functions, 235–239
  - control characters in quoted strings, 237–238
  - displaying messages, 236
  - wild-card matching, 238–239
- overview, 2
- predefined variables, 232–233
- program files, 229–231
  - color coding, 231
  - comments, 230
  - creating new source files, 229
  - editing code in VLISP, 229

## AutoLISP (continued)

- formatting code, 229–230
- property-handling functions, 460
- protected symbols, 94, 228–229
- query and command functions, 260–264
  - configuration control, 264
  - foreign-language support, 261
  - inspecting and changing system and environment variables, 263–264
- passing pick points to AutoCAD
  - commands, 262–263
- pausing for user input, 261–262
- sending commands to AutoCAD
  - prompt, 260–261
- synopsis of, 451
- undoing commands issued with
  - command function, 263
- reactor functions, 464–466
- referring to entities for multiple sessions, 226
- relationship to Visual LISP, 1, 3
- running from Console prompt, 20–21
- running selected lines of code, 21
- selection set handling functions, 296–305
  - adding entities, 303–304
  - creating selection sets, 296–297
  - deleting entities, 303, 304
  - finding number of entities, 304
  - passing selection sets between AutoLISP and ObjectARX
    - applications, 305
  - returning entity names, 304
  - selection set filter lists, 298–304
  - synopsis of, 455
  - testing whether an entity is a member, 304
- selection set, object
  - and symbol table functions, 453–457
- spaces in code, 229–230
- special forms, 253–254
- string handling, 234–235, 444–445
- symbol and function-handling, 245–254
- symbol table access functions, 332–333
- symbol handling, 245, 445
- undoing commands issued with command function, 263
- user input functions
  - accessing user input from devices, 291
  - allowable input, 268–269
  - arbitrary keyboard input, 273
  - controlling user-input function conditions, 271–273
  - getting user input, 268–271
  - getxxx functions, 268–271
  - input options, 271–272
  - keyword options, 272–273

## AutoLISP (continued)

- pausing for user input, 261–262
  - synopsis of, 452
  - validating input, 273
  - utility functions, 446–452
    - conversion functions, 446
    - device access functions, 447
    - display-control functions, 448
    - file-handling functions, 449
    - geometric functions, 450
    - query and command functions, 451
    - user input functions, 452
  - variables
    - assigning values to, 227–228, 231
    - data type, 227–229, 231
    - displaying value of, 231–232
    - nil variables, 232
    - predefined, 232–233
  - Visual LISP extended functions, 22, 457
    - collection manipulation functions, 458
    - curve measurement functions, 461–462
    - data conversion functions, 458–459
    - defined, 22
    - dictionary functions, 462
    - loading extended functions, 22
    - method invocation functions, 459
    - object-handling functions, 463
    - property-handling functions, 460
    - running, 22
    - vl-load-com and, 22
    - VLX namespace functions, 467
  - Windows Registry functions, 469
  - xrecord objects, 331
  - AutoLISP Format Options command (VLISP Tools menu), 51–59
  - AutoLISP functions. *See* functions
  - AutoLISP Reference, 4, 205, 208
  - automatically backing up text editor files, 31–32
  - :AXVLO-IO-CALLBACK trace keyword frame, 90
- ## B
- backing up
    - files in text editor automatically, 31–32
    - restoring from backup files, 32
  - backslash (\)
    - for control characters in quoted strings, 237–238
    - using within quoted strings, 237
  - balance of parentheses, checking in VLISP, 61–65, 222–223
  - base.dcl file
    - default\_button definition, 342
    - overview, 340
  - beeping your PC, 206
  - :BEFORE-EXP trace keyword frame, 90

- big\_increment (DCL attribute), 398
- blackboard namespace, 133
- blackboard variables, 133, 134
- blocks
  - viewing with VLISP, 106–107
  - working with, 319
- blocks in the drawing database, 106–107
- blocks of text
  - converting code to comments, 60
  - converting comments to active text, 60
  - indenting or unindenting, 59
  - saving to new file, 36, 60
  - sorting, 60
- bold courier type in this manual, 7
- boldface type in this manual, 7
- bookmarking text, 40–41
  - adding bookmarks, 40
  - clearing, 41
  - deleting bookmarks, 41
  - moving to bookmarks, 40–41
  - overview, 40
  - selecting text using bookmarks, 41
- boxed\_column tile, 410–411
- boxed\_radio\_column tile, 411
- boxed\_radio\_row tile, 412
- boxed\_row tile, 412
- break loops
  - Break on Error option, 75
  - Break on Function Entry, 75, 94
  - continuable break loops, 77–78
  - continuing program execution, 74, 77
  - Debug-on-Entry flag, 75, 94
  - entering, 75–77
  - exiting, 74, 77
  - non-continuable break loops, 78
  - overview of, 76
  - Prompt to enter break loop option for protected symbols, 228
  - Quit Current Level option, 77
  - stepping through programs in, 71–72, 78
  - turning on, 75–77
- Break on Error command (VLISP Debug menu), 75
- Break on Function Entry command (VLISP Debug menu), 75
- Breakpoint Service dialog box, 80
- :BREAK-POINT trace keyword frame, 90
- breakpoints
  - clearing, 79
  - continuing program execution, 74, 77
  - Debug-on-Entry flag, 75, 94
  - described, 68
  - disabling temporarily, 80
  - highlight colors for, 79
  - life cycle of, 81
  - listing and viewing, 80
- breakpoints (*continued*)
  - monitoring evaluation results of an expression, 73–74
  - Prompt to enter break loop option for protected symbols, 228
  - removing, 79
  - Reset to Top Level option, 77, 92
  - resuming execution from, 77
  - setting, 70–71, 79
  - stepping through programs from, 71–72, 78
  - Stop Once mode, 75, 76
  - Toggle Breakpoint command, 33
  - turning on, 75–77
- Breakpoints window, 80
- Browse All Entities command (VLISP View menu), 104
- Browse Blocks command (VLISP View menu), 106
- Browse Selection command (VLISP View menu), 107
- Browse Tables command (VLISP View menu), 105
- browsing. *See* displaying
- Build Options dialog box, 154
- Build Output window
  - error messages in, 65
  - finding location of syntax errors, 65
  - Make Application output, 123
- Build Project FAS button (Project window), 145, 148
- building applications, 109–136
  - compiling and linking programs, 110–115
    - compile example, 113–114
    - compiling programs from files, 111–113
    - linking function calls, 111, 116, 159
    - loading and running compiled programs, 114–115
    - using the compiler, 110
  - designing for multiple document environments, 126–129
    - error-handling in MDI environment, 134–135
    - namespaces overview, 126–128
    - running an application in its own namespace, 129–130
    - sharing data between namespaces, 133–134
- making application modules, 117
  - changing application options, 124–125
  - including projects, 151–152
  - loading and running VLISP applications, 124
  - rebuilding applications, 125
  - updating applications, 125, 126

Built-in function (Apropos filter value), 44  
button tile, 413  
buttons (dialog box). *See* tiles (dialog box)

## C

.c files, 140  
c:xxx functions, 247–250  
caddr function, 243  
cadr function, 243, 244  
calibrating tablets, 291–293  
Call Point Source command (Trace Stack window shortcut menu), 88  
call stack, 68  
callback events for reactors, 204–205  
callback functions for dialog boxes. *See* dialog box management, action expressions and callbacks  
callback functions for reactors, 205–207  
calling ActiveX methods, 164–173, 198  
capitalization. *See* case  
Capitalize option, 60  
car function  
    handling dotted pairs, 244  
    for point lists, 243, 244  
    retrieving items from lists, 241  
case (of text and symbols)  
    Assign-Protect flag and, 57  
    automatic changing of by AutoLISP, 57, 234  
    conversion by Apropos, 48  
    converting selected text, 60  
    converting with strcase function, 234  
    DCL attributes and, 393  
    dialog box design guidelines, 356–357  
    equality checking and, 240  
    symbols, automatic case changing of, 57, 234  
    tile names, 342  
case sensitivity  
    of Apropos search, 48  
    comparison functions and, 240  
    of Complete Word by Match, 48  
    of DCL attribute names and values, 344  
    of DCL tile names, 343  
    of equality functions, 240  
    of grouping operators, 302  
    of input functions, 272  
    of mnemonics (in DCL tiles), 405  
    of reserved words (in DCL tiles), 392  
    of search in text editor, 38  
    of symbol names, 227  
.ch files, 140  
cdr function  
    handling dotted pairs, 244  
    returning all but first list element, 241  
changing X axis of circle, 181–183  
Check command, 64–65  
    errors detected by, 64–65

Check command (*continued*)  
    finding location of errors, 65  
    running, 65  
Check Selection command (VLISP Tools menu), 65  
Check Syntax command (Project window shortcut menu), 146  
Check Text in Editor command (VLISP Tools menu), 65  
checking for syntax errors, 61–65  
    balance of parentheses, 61–63  
    from Project window, 146  
    using Check command, 64–65  
    using color coding, 63  
children\_alignment (DCL attribute), 398  
children\_fixed\_height (DCL attribute), 398–399  
children\_fixed\_width (DCL attribute), 399  
circles  
    changing X axis, 181–183  
    returning minimum and maximum bounding points, 184–185  
Clear All Bookmarks command (VLISP Search menu), 41  
Clear All Breakpoints command (VLISP Debug menu), 79  
Clear Console window command (Console window shortcut menu), 29  
clear language in dialog box design, 353  
clearing. *See* deleting  
client\_data\_tile function, 386  
Clipboard  
    Console window shortcut menu commands, 29  
    text editor shortcut menu commands, 32–33  
    VLISP support for, 36  
    *See also* copying; moving text  
close function, 227  
Close Parenthesis Style options (Format Options dialog box), 54  
Close Project command (Project window shortcut menu), 146  
closing  
    all Inspect windows, 103  
    AutoCAD, 22  
    Console log file, 30  
    dialog box closing functions, 427  
    files in AutoLISP programs, 227  
    projects, 149  
    Visual LISP, 22  
coding aids  
    Apropos feature, 43–47  
    color coding, 19, 42–43, 63  
    Complete Word by Apropos feature, 48–49  
    Complete Word by Match feature, 47–48  
    formatting code, 50–60  
    *See also* formatting text

- collection objects
  - applying function to every object in, 187–188
  - collection manipulation functions, 458
  - evaluating series of functions with each object in, 187, 187–188
  - overview, 164
  - retrieving member objects, 188–189
  - working with, 187–189
- color (DCL attribute), 399–400
- color coding
  - for AutoLISP code, 231
  - changing colors, 42
  - of DCL files, 345
  - of files in text editor, 19, 42–43
  - scheme for AutoLISP code, 42
  - for syntax error detection, 63
- colors for breakpoints, 79
- Column style, 53
- column tile, 413–414
- combining lists, 241
- combining strings in AutoLISP, 234
- command function
  - foreign-language support, 261
  - passing pick points to AutoCAD commands, 262–263
  - pausing for user input, 261–262
  - sending commands to AutoCAD prompt, 260–261
  - undoing commands, 263
- command prompt, starting Visual LISP from, 12
- commands
  - Console window shortcut menu commands, 28
  - descriptions on status bar, 14
  - interrupting, 27
  - issuing from Console window, 14
  - issuing from VLISP menus, 15
  - retrieving previous commands, 18, 26–27
  - text editor shortcut menu commands, 32–33
  - See also* specific commands
- Comment Block option, 60
- comments
  - in AutoLISP program files, 230
  - in dialog control language (DCL), 344
  - Insert Form-Closing Comment option, 55
  - shortcut keys for, 60
  - Split Comments option, 56
  - in unit definition file, 286
  - VLISP comment styles, 58
- Compilation mode option (Build Options dialog box), 154
- compilation report detail level, 155
- compiling and linking programs
  - Application Compilation Options dialog box, 122–123
  - compiling and linking programs (*continued*)
    - build options, choosing, 144, 153–155
    - choosing a compilation mode, 111, 154, 155
    - compile example, 113–114
    - compiling programs from files, 111–113
    - defining build options, 153–155
    - identifying the input file, 112
    - linking function calls, 111, 116, 159
    - loading and running compiled programs, 114–115
    - localizing variables, 155
    - Make Application output, 123
    - Merge Files mode, 154
    - optimization during compiling, 111
    - output file, naming, 113
    - project source files, 145–148
    - recompiling project source files, 138, 145–146, 148
    - report detail level, 155
    - safe optimization, 155, 157–160
    - source files, identifying, 112–113
    - using Application Wizard, 117
    - using the compiler, 110
    - using VLISP integrated project management facilities, 110
    - vlisp-compile function, 111–113
    - See also* Application Wizard; optimizing application code
  - Complete Word by Apropos feature, 48–49
  - Complete Word by Match feature, 47–48
  - concatenating strings in AutoLISP, 234
  - concatenation tile, 414
  - COND-expression clauses, Column style for, 53
  - conditional branching and looping in AutoLISP, 240
  - configuration control, 264
  - Configure Current command (VLISP Tools menu), 42, 79
  - connecting with ActiveX
    - to application objects, 165
    - to applications, 195
- cons function
  - adding items to list beginning, 241, 241
  - creating dotted pairs, 244
- consistency in dialog box design, 353
- Console prompt
  - changing active document, 28
  - clearing, 18, 27
  - copying symbol name from Apropos window, 46
  - copying text within the Console window, 25
  - displaying new prompt without evaluating text, 18, 27
  - displaying variable values, 231–232
  - illustrated, 25



- Console prompt (*continued*)
  - pasting text at, 25
  - resetting, 18, 27
  - running AutoLISP programs from, 20–21
- Console window
  - AutoCAD command window compared to, 18, 25–26, 28
  - changing active document, 28
  - clearing Console prompt, 18
  - color coding of text, 42
  - Complete Word by Apropos feature, 48–49
  - Complete Word by Match feature, 47–48
  - continuing expression on new line, 25
  - copying text to the prompt, 25
  - displaying new prompt without evaluating text, 18, 27
  - ENTER key and, 25–26
  - entering multiple expressions before processing, 25
  - features, 18–19
  - interactive programming in, 25
  - interrupting commands, 27
  - loading AutoLISP programs from text editor, 20
  - logging activity, 29–30
  - Make Application output, 123
  - multiple AutoCAD drawings with, 28
  - overview, 14, 18–19, 25–26
  - printing Frame Binding window value to, 88
  - printing Inspect window object to, 102
  - printing stack element to, 88
  - printing Watch window variable values, 85
  - retrieving previous commands, 18, 28
  - running AutoLISP programs, 20–21
  - searching input history, 18, 27
  - shortcut menu commands, 28
- context-sensitive help. *See* help feature
- continuable break loops, 77–78
- Continue command (VLISP Debug menu), 74, 77
- continuing expression on new line in Console window, 25
- control characters
  - in DCL strings, 392
  - in quoted strings, 237–238
- controlling amount of Inspect information, 104
- controlling AutoCAD display, 264–268
  - graphics and text windows, 267
  - low-level graphics, 267–268
  - menus, 264–268
  - overview, 264
- conversion functions, 279–289
  - angular values from radians or degrees, 281–282
  - ASCII code conversions, 282–283
- conversion functions (*continued*)
  - coordinate system transformations, 286–289
    - AutoCAD coordinate systems, 286–287
    - overview, 286
    - point transformations, 289
    - specifying coordinate systems, 288
    - valid integer codes, 288
  - measurement unit conversions, 283–286
  - point transformations, 289
  - string conversions, 279–281
  - synopsis of, 446
- converting AutoLISP data types to ActiveX data types, 173–180
  - ActiveX and VBA Reference, 173–174
  - AutoLISP data types accepted in place of ActiveX data type, 180
  - other AutoLISP data types, 179–180
  - using safearrays with variants, 178–179
  - working with safearrays, 175–178
  - working with variants, 174–175
- converting object references, 190–191
  - enames and VLA-objects, 190
  - obtaining one object identifier from another, 190–191
- converting string case, 60, 234
- coordinate system transformations
  - AutoCAD coordinate systems, 286–287
  - entity context and coordinate transform data, 307–311
  - overview, 286
  - point transformations, 289
  - specifying coordinate systems, 288
  - valid integer codes, 288
- Copy command
  - Apropos results window shortcut menu, 46
  - Console window shortcut menu, 29
  - Frame Binding window shortcut menu, 89
  - text editor shortcut menu, 33
  - Trace Stack window shortcut menu, 88
- Copy to Clipboard command (Apropos results window shortcut menu), 46
- Copy to Trace/Log feature (Apropos window), 46
- Copy Value command (Watch window shortcut menu), 84
- copying
  - Apropos search results to Trace window and log, 46
  - by dragging in text editor, 36
  - Clipboard support for, 36
  - from Console window, 25
  - from Frame Binding window to \*obj\* system variable, 89
  - from Inspect window to \*obj\* system variable, 103

## copying (*continued*)

- from Inspect window to Trace window and log, 102
  - from output windows, 15
  - selected text to new file, 36, 60
  - stack elements to \*obj\* system variable, 88
  - symbol names from Apropos results window, 46
  - symbols from Apropos results window, 46
  - undoing copy actions, 37
  - using Console window shortcut menu, 29
  - using text editor shortcut menu, 33
  - from Watch window to \*obj\* system variable, 84
  - from Watch window to Trace window and log, 84
- See also* printing
- correcting text. *See* text editor
- Courier type in this manual, 7
- .cpp files, 140
- creating
- applications, 117–123
  - complex entities, 316–318
  - Console log file, 30
  - new instance of application object, 195
  - new source files, 31, 229
  - reactors, 207–210
  - safearrays, 175–177
  - selection sets, 296–297
  - variant with an array of four doubles, 179
  - variants, 174, 174–175
- Current File option (Find dialog box), 37
- Current Selection option (Find dialog box), 37
- curve measurement functions, 461–462
- curve-fit polylines, processing, 321
- Customization Guide, 8
- Cut command, text editor shortcut menu, 33
- cutting. *See* deleting; moving
- cvunit function, 283–284

## D

data inspection tools. *See* Frame Binding window; Inspect feature; Symbol Service dialog box; Trace Stack window

## data types

### AutoLISP

- accepted in place of ActiveX data type, 180
- converting to ActiveX data types, 173–180
- entity names, 226
- file descriptors, 226–227
- integers, 224–225
- lists, 225
- overview, 224–229
- reals, 225
- selection sets, 226

## data types (*continued*)

- strings, 225
  - symbols, 227–229
  - variables, 227–229, 231
  - VLA-objects, 226
- converting AutoLISP data types to ActiveX data types, 173–180
- ActiveX and VBA Reference, using, 173–174
  - other AutoLISP data types, 179–180
  - using safearrays with variants, 178–179
  - working with safearrays, 175–178
  - working with variants, 174–175
- element list formats and, 97–102
- extended data organization, 324–326
- for tile attribute values, 392
- Database reactors, 202
- date
- changing format for, 60
  - inserting current date, 60
- DCL (dialog control language)
- acad.dcl* file, 340
  - actions, 372
  - adjusting layout, 347–352
    - centering buttons, 347–348
    - customizing exit button text, 351–352
    - distributing tiles in a cluster, 348–349
    - example, 347–348
    - space around boxed row or column, 350–351
    - space at right side or bottom, 350
    - space between tiles, 349
- attributes, 392–394
- case-sensitivity, 393
  - data types for values, 392
  - naming, 393
  - numeric values and, 393
  - overview, 392–394
  - restricted attributes, 393
  - synopsis of predefined attributes, 394–396
  - syntax, 343–344
  - user-defined attributes, 393–394
- attributes listed by function
- action expression specification, 397
  - aligning horizontally or vertically in a cluster, 397–426
  - aspect ratio specification, 398
  - background color of image, 399–400
  - bold text specification, 402
  - button is selected with ESC key (or not), 402
  - callback specification, 397
  - character used to mask user input, 406
  - character width units for edit box input, 400

## DCL (dialog control language)

- default alignment for all tiles in a cluster, 398
- default button specification, 402–403
- default height for all tiles in a cluster, 398–399
- default width for all tiles in a cluster, 399
- fixed pitch font for list boxes or pop-up lists, 401
- initial availability of tile, 403
- initial choices for list box or pop-up list, 404
- initial value of tile, 407
- key of tile to receive initial keyboard focus, 402
- keyboard mnemonic character for tile, 405
- maximum characters for edit box input, 400
- maximum value for slider, 405
- minimum value for slider, 405
- multiple selections allowed in list box (or not), 406
- slider incremental control value specification, 398, 406
- slider orientation specification, 404
- specifying if tile is activated, 398
- tab placement in character width units, 406
- text larger than tab stop truncated in list box or pop-up list (or not), 407
- tile height allowed to fill space (or not), 401
- tile height specification, 401–402
- tile name specification, 403
- tile receives focus with TAB key (or not), 403
- tile text (label) specification, 404
- tile width allowed to fill space (or not), 401
- tile width specification, 407
- attributes listed by name, 397–407
  - action, 397
  - alignment, 397–426
  - allow\_accept, 398
  - aspect\_ratio, 398
  - big\_increment, 398
  - children\_alignment, 398
  - children\_fixed\_height, 398–399
  - children\_fixed\_width, 399
  - color, 399–400
  - edit\_limit, 400
  - edit\_width, 400
  - fixed\_height, 401
  - fixed\_width, 401

## DCL (*continued*)

- fixed\_width\_font, 401
- height, 401–402
- initial\_focus, 402
- is\_bold, 402
- is\_cancel, 402
- is\_default, 402–403
- is\_enabled, 403
- is\_tab\_stop, 403
- key, 403
- label, 404
- layout, 404
- list, 404
- max\_value, 405
- min\_value, 405
- mnemonic, 405
- multiple\_select, 406
- password\_char, 406
- small\_increment, 406
- tab\_truncate, 407
- tabs, 406
- value, 407
- width, 407
- base.dcl* file, 340
- control characters in DCL strings, 392
- DCL files, 340–341
- DCL syntax, 341–344
  - attributes and attribute values, 343–344
  - comments, 344
  - semantic auditing of DCL files, 346–347
  - tile definitions, 342
  - tile references, 343
- defining dialog boxes, 340–344
  - DCL syntax, 341–344
  - referencing DCL files, 340
- displaying dialog boxes with VLISP, 344–347
  - color coding in text editor, 345
  - example, 344
  - preview error handling, 346
  - selecting a dialog box from multiple definitions, 342
  - semantic auditing of DCL files, 346–347, 387–388
- error handling
  - alert boxes, 355, 364
  - error tile guidelines, 363–364
  - forgiving errors, 355
  - preview error handling, 346
  - semantic auditing of DCL files, 346–347
- referencing DCL files, 340
- symbolic names for colors, 399–400
- tiles in functional groupings, 408–410
  - decorative and informative tiles, 409

## DCL (*continued*)

- dialog box exit buttons and error tile, 410
- predefined active tiles, 408
- restricted tiles, 410
- text clusters, 409
- tile clusters, 408
- tiles listed by name, 410–426
  - boxed\_column, 410–411
  - boxed\_radio\_column, 411
  - boxed\_radio\_row, 412
  - boxed\_row, 412
  - button, 413
  - column, 413–414
  - concatenation, 414
  - DCL (dialog control language), 426
  - dialog, 414
  - edit\_box, 415
  - errtile, 415
  - image, 416
  - image\_button, 416–417
  - list\_box, 417
  - ok\_cancel, 418
  - ok\_cancel\_help, 418
  - ok\_cancel\_help\_errtile, 418–419
  - ok\_cancel\_help\_info, 419
  - ok\_only, 418
  - paragraph, 419–420
  - popup\_list, 420–421
  - radio\_button, 421
  - radio\_column, 422
  - radio\_row, 422
  - row, 423
  - slider, 423
  - spacer, 425
  - spacer\_0, 426
  - text, 424
  - text\_part, 424
  - toggle, 425
- DCL files
  - acad.dcl*, 340
  - base.dcl*, 340
  - color coding, 345
  - overview, 340–341
  - parts of, 340
  - referencing, 340
  - semantic auditing of, 346–347, 387–388
  - specifying in VLISP, 344
- :DCL-ACTION trace keyword frame, 89
- De symbol flag, 94
- Debug menu (VLISP)
  - Animate command, 74
  - Break on Error command, 75
  - Break on Function Entry command, 75, 94
  - Clear All Breakpoints command, 79
  - Continue command, 74, 77
  - Debug Top-Level mode command, 75

## Debug menu (VLISP)

- overview, 17
- Quit Current Level command, 77
- Reset to Top Level command, 77, 92
- Step Into command, 71, 78
- Step Out command, 78
- Step Over command, 72, 78
- Stop Once command, 75, 76
- Toggle Breakpoint command, 70, 79
- Debug toolbar
  - Continue button, 77
  - Quit Current Level button, 77
  - Reset to Top Level button, 77, 92
  - Step Indicator button, 71
  - Step Into button, 71, 78
  - Step Out button, 78
  - Step Over button, 72, 78
  - Toggle Breakpoint button, 70, 79
- Debug Top-Level mode command (VLISP Debug menu), 75
- debugging programs
  - break loop process, 76
  - Break on Error option, 75
  - breakpoint life cycle, 81
  - breakpoints
    - clearing, 79
    - continuing program execution, 74, 77
    - Debug-on-Entry flag, 75, 94
    - described, 68
    - disabling temporarily, 80
    - duration of, 81
    - highlight colors for, 79
    - life cycle of, 81
    - listing and viewing, 80
    - monitoring evaluation results of an expression, 73–74
    - Prompt to enter break loop option for protected symbols, 228
    - removing, 79
    - Reset to Top Level option, 77
    - resuming execution from, 77
    - setting, 70–71, 79
    - stepping through programs from, 71–72, 78
    - Stop Once mode, 75, 76
    - turning on, 75–77
  - checking for syntax errors, 61–65
  - continuable break loops, 77–78
  - continuing program execution, 74, 77
  - Debug-on-Entry flag, 75, 94
  - disabling breakpoints temporarily, 80
  - displaying breakpoints, 80
  - example, 69–75
  - highlight colors for breakpoints, 79
  - listing breakpoints, 80
  - monitoring evaluation results of an expression, 73–74

- debugging programs (*continued*)
  - non-continuable break loops, 78
  - optimization and bug introduction, 156
  - Quit Current Level option, 77
  - removing breakpoints, 79
  - Reset to Top Level option, 77, 92
  - running in Animate mode, 74
  - semantic auditing of DCL files, 346–347, 387–388
  - setting breakpoints, 70–71
  - starting debugging sessions, 76
  - starting Trace logging, 82
  - stepping through programs from breakpoints, 71–72, 78
  - Stop Once mode, 75, 76
  - Top-Level debugging mode, 75
  - using Frame Binding window, 88–92
  - using Inspect windows, 95–108
  - using Symbol Service dialog box, 92–95
  - using Trace Stack window, 85–92
  - using Watch window, 83–85
  - VLISP debugging features, 68–69
  - See also* break loop mode; checking for syntax errors; errors; Frame Binding window; Inspect feature; Symbol Service dialog box; Trace Stack window
- Debug-on-Entry symbol flag, 75, 94
- defining callback functions, 205–207
- defining functions. *See* defun function
- defining new units, 284
- defining projects, 140–149
  - assigning properties, 141–144
  - overview, 140
  - sample files, 140
  - using the Project window, 145–148
- defun function, 245–253
  - adding commands, 248–249
  - c:xxx functions, 247–250
  - compatibility with AutoCAD versions, 246–247
  - defining functions, 245–247
    - with arguments, 252–254
  - local variables in functions, 251–252
  - redefining AutoCAD commands, 249–250
- defun-q function, 281–282
- degrees, converting to radians, 282
- Delete Blanks option, 60
- Delete to EOL option, 60
- deleting
  - bookmarks, 41
  - breakpoints, 79
  - clearing Console prompt, 18
  - determining if objects were erased, 183
  - disabling reactors, 214–215
  - entities, 311
  - project members, 145–146
- deleting (*continued*)
  - reactor owner objects, 214
  - selection set entities, 303, 304
  - spaces from cursor to first non-blank character, 60
  - stripping file extensions, 234–235
  - text from cursor to end of line, 60
  - trailing spaces, 36
  - trailing tab characters, 36
  - using text editor shortcut keys, 33
  - variables from Watch window, 84, 85
- derived units, 285–286
- Descend command (Inspect window shortcut menu), 102
- designing dialog boxes. *See* dialog box design
- designing for multiple document environments
  - AutoLISP limitations, 136
  - error-handling in MDI environment, 134, 135
  - limitations using AutoLISP, 136
  - namespaces overview, 126–128
  - running an application in its own namespace, 129–130
  - sharing data between namespaces, 133–134
- detecting syntax errors. *See* checking for syntax errors
- developing programs, 24–65
  - basic steps, 24
  - checking for syntax errors, 61–65
  - formatting code, 50–60
  - using coding aids, 42–50
  - using the Console window, 25–30
  - using the text editor, 31
- device access and control functions, 291–293
  - accessing user input, 291
  - calibrating tablets, 291–293
  - synopsis of, 447
- dialog box design, 337–364
  - acad.dcl file, 340
  - adjusting layout, 347–352
    - centering buttons, 347–348
    - customizing exit button text, 351–352
    - distributing tiles in a cluster, 348–349
    - example, 347–348
    - layout guidelines, 357
    - space around boxed row or column, 350–351
    - space at right side or bottom, 350
    - space between tiles, 349
  - base.dcl file, 340
  - defining dialog boxes, 340–344
    - DCL syntax, 341–344
    - referencing DCL files, 340
    - tile definitions, 341–342
  - dialog box components, 338–339
  - displaying dialog boxes with VLISP, 344–347

## dialog box design (*continued*)

- color coding in text editor, 345
- example, 344
- preview error handling, 346
- selecting a dialog box from multiple definitions, 342
- semantic auditing of DCL files, 346–347, 387–388
- error handling
  - alert boxes, 355, 364
  - error tile guidelines, 363–364
  - forgiving errors, 355
  - preview error handling, 346
  - semantic auditing of DCL files, 346–347, 387–388
- guidelines, 353–364
  - abbreviations, avoiding, 357
  - capitalization, 356–357
  - consistency, 353
  - disabling tiles, 358
  - error handling, 363–364
  - forgiving errors, 355
  - handling keyboard input, 359
  - hiding dialog boxes, 358
  - international language considerations, 359–360
  - layout, 357
  - nesting dialog boxes, 358
  - placement of dialog boxes, 357
  - predefined tiles and clusters, 360–364
  - providing defaults, 359
  - providing help, 355
  - size of dialog boxes, 357
  - user control over input, 353–354
  - for users with disabilities, 356
- planning the design, 339
- predefined tiles and clusters, 360–364
  - buttons, 360
  - clusters, 361
  - edit boxes, 361
  - image buttons and image tiles, 361–362
  - list boxes, 362
  - radio buttons, radio rows, and radio columns, 360–362
  - sliders, 362
  - text, 363
  - toggles, 363
- user input
  - handling keyboard input, 359
  - user control over, 353–354
  - for users with disabilities, 356
- dialog box management, 365–389
  - action expressions and callbacks
    - for application-specific data, 386
  - callback reasons, 370–372
  - DCL actions, 372

## dialog box management (*continued*)

- default actions, 372
- for edit boxes, 376
- hiding dialog boxes, 377–378
- for image buttons, 385
- nesting dialog boxes, 376–377
- overview, 368–372
- processing list elements, 382
- for radio clusters, 374–375
- for sliders, 375–376
- variables, 369–370
- application-specific data, 386
- functions handling, 429
- callback reasons, 370–372
  - edit boxes, 371
  - image buttons, 372
  - list boxes, 371–372
  - reason codes, 371
  - sliders, 371
- canceling dialog boxes
  - all dialog boxes, 368, 377, 378
  - if Cancel button and Default button the same, 352
  - hiding dialog boxes and, 377
  - ok\_cancel tiles, 343
- controlling dialog boxes with AutoLISP programs, 366–368
  - displaying the dialog box and responding to the user, 366–367
  - functions restricted when dialog box is open, 367–368
  - overview, 366–367
- DCL error handling, 386–388
  - auditing level for error messages, 387–388
  - for multiple DCL files, 387
  - testing dialog boxes with programs, 387
  - when loading first time, 386–387
- dialog box opening and closing functions, 427
- displaying the dialog box and responding to the user, 366–367
- function sequence, 388–389
- function synopsis, 427–429
  - application-specific data handling functions, 429
  - dialog box opening and closing functions, 427
  - image tile handling functions, 429
  - list box and pop-up list handling functions, 428
  - tile and attribute handling functions, 428
- functions restricted when dialog box is open, 367–368

- dialog box management (*continued*)
  - handling tiles, 372–376
    - changing modes and values at callback time, 373–374
  - edit boxes, 376
  - initializing modes and values, 372–373
  - mode codes for mode\_tile function, 373
  - radio clusters, 374–375
  - sliders, 375–376
- hiding dialog boxes, 377–379
  - canceling all dialog boxes, 378
  - requesting a password, 379
  - user input and, 377–378
- image tiles and buttons, 383–385
  - color attributes, 384
  - creating images, 383–385
  - handling image buttons, 385
  - image tile handling functions, 429
- list boxes and pop-up lists, 380–382
  - appending list items, 381
  - changing list items, 381
  - creating lists, 380–381
  - deleting list items, 381
  - functions handling, 380, 428
  - inserting list items, 381
  - list operations, 380–381
  - operation codes for start\_list function, 380
  - processing list elements, 382
- nesting dialog boxes
  - canceling all dialog boxes, 368, 377, 378
  - managing, 376–377
  - overview, 366–367
  - requesting passwords from users, 379
  - sample block definition dialog box, 389
- dialog boxes
  - all dialog boxes, 368, 377, 378
  - if Cancel and Default button are the same, 352
  - hiding dialog boxes and, 377
  - ok\_cancel tiles, 343
- dialog control language. *See* DCL
- dialog tile, 414
- dictionary and symbol table handling
  - functions, 455–456
- dictionary functions, 334, 462
- dictionary objects, 323
- dictnext function, 334
- dictsearch function, 334
- DIESEL string expressions, 266–267
- digitizing tablets, calibrating, 291–293
- directories
  - application directory, 118
  - for Console log file, 30
- directories (*continued*)
  - for .fas files, 154
  - Make Application output, 123
  - project home directory, 141
  - project source directory, 141
  - searching for text in, 38
  - Tmp directory for project files, 154
- disabling
  - breakpoints temporarily, 80
  - reactors, 214–215
- display control functions, 448
- Display Coordinate System, 287
- displaying
  - AutoCAD drawing entities, 104–108
  - breakpoints, 80
  - controlling AutoCAD display, 264–268
    - graphics and text windows, 267
    - low-level graphics, 267–268
    - menus, 264–268
    - overview, 264
  - current trace stack, 87
  - current value of variables, 18, 25
  - dialog boxes with VLISP, 344–347
    - color coding in text editor, 345
    - example, 344
    - preview error handling, 346
    - selecting a dialog box from multiple definitions, 342
    - semantic auditing of DCL files, 346–347, 387–388
  - display control functions, 448
  - error trace stack, 92
  - Frame Binding window for local variables, 88
  - function call stack, 68
  - function definition for Symbol Service
    - dialog box, 94
  - messages in AutoLISP, 236
  - new Console prompt without evaluating text, 18, 27
  - position of caller expression in Trace Stack window, 88
  - shortcut menus, 28
  - source text for functions, 88
  - trace stack element information, 87–88
  - variable values, 231–232
  - See also* Inspect feature; printing
- dlistof function, 280
- Document object, 168
- Document reactors, 202
- dollar sign (\$) prompt. *See* Console prompt
- done\_dialog function
  - hiding dialog boxes and, 377
  - nested dialog boxes and, 377
  - for OK button, 366
  - start\_dialog calls and, 367

- done\_dialog function (*continued*)
  - user input based on graphics screen and, 368
  - Windows and, 367
- dotted pairs, 244–245
- Down option (Find dialog box), 37
- Downcase option, 60
- Downcase Symbols option (Apropos dialog box), 44
- dragging text in text editor, 36
- drawings
  - adding entities to, 315–316
  - AutoCAD drawing database
    - viewing blocks in, 106–107
    - viewing entities in, 104–105
    - viewing extended data, 108
    - viewing selected objects in drawings, 107
    - viewing symbol tables in, 105
  - multiple AutoCAD drawings with Console window, 28
  - opening drawings containing persistent reactors, 215–216
  - viewing drawing entities, 104–108
    - blocks in the drawing database, 106–107
    - controlling amount of Inspect information, 104
  - entities in the drawing database, 104–105
  - extended data associated with objects, 105, 108
  - selected objects in drawings, 107
  - shortcut menu for object line, 105
  - symbol tables in the drawing database, 105
- drawline function
  - loading from text editor, 20
  - running from Console prompt, 20–21
- Drop directives and Safe Optimization, 158, 160
- .dsk files, 139
- :DWF trace keyword frame, 90
- :DWG trace keyword frame, 90
- DXF codes. *See* DXF Reference
- DXF object list (ENAME Inspect window), 101
- DXF Reference, 8

## E

- Ea symbol flag, 95
- edit boxes. *See* tiles (dialog box)
- Edit command (Project window shortcut menu), 146
- Edit Global Declarations option (Build Options dialog box), 154
- Edit menu (VLISP)
  - Match Backward command, 61–63
  - Match Forward command, 61–63

- Edit menu (*continued*)
  - overview, 16
  - Redo command, 31
  - Select Backward command, 62
  - Select Forward command, 62
  - Undo command, 31
- Edit toolbar
  - New File button, 31
  - Open button, 32
  - Redo button, 31
  - Undo button, 31
- edit\_box tile, 415
- edit\_limit (DCL attribute), 400
- edit\_width (DCL attribute), 400
- editing. *See* text editor
- Editor reactors, 203–204
- enabling reactors, 215
- ENAME Inspect window, 101
- enames
  - converting to and from VLA-objects, 190
  - searching for associated handle, 190
- end\_image function, 383
- end\_list function, 381
- entdel function, 311
- ENTER key and Console window processing, 25–26
- entget function, 165, 311–313, 323, 327–328
- entities
  - adding to drawings, 315–316
  - blocks, 319
  - changing, 314–315
  - complex, 311, 316
  - handles and use of, 307
  - modifying, 314–315
  - names, obtaining, 305–306
  - obtaining information on, 311–313
  - referring to across multiple sessions, 226
  - viewing in drawing database, 104–105
- entity access functions, 311
- entity data functions, 311–320
  - adding entities to drawings, 315–316
  - anonymous blocks, 319
  - creating complex entities, 316–318
  - deleting entities, 311
  - graphics screen and, 319–320
  - modifying entities, 314–315
  - obtaining entity information, 311–313
  - working with blocks, 319
- entity filter lists for selection sets, 298–304
  - examples, 298–299
  - filtering for extended data, 300
  - logical grouping of filter tests, 302–303
  - overview, 298–300
  - relational tests, 300–302
  - selection set manipulation, 303–304
  - wild-card patterns in, 300



- entity name data type
  - defined, 226
  - overview, 226
  - referring to entities across multiple sessions, 226
- entity name functions, 305–311
  - entity access functions, 311
  - entity context and coordinate transform data, 307–311
  - entity handles and their uses, 307
  - overview, 305–306
  - retrieving entity names, 305–306
  - setting entity name to variable name, 306
- entlast function, 306
- entmake function, 315–319, 321–322
- entmod function
  - dictionary objects and, 323
  - graphics screen and, 319
  - modifying entities, 314–315
  - non-graphic object handling, 322
  - polylines and, 321
  - symbol table objects and, 322
- entnext function, 306, 321
- :ENTRY-NAMESPACE trace keyword frame, 90
- entsel function, 305–306
- entupd function, 319–320
- environment variables, inspecting and changing, 264
- equality and conditional functions, 439
- equality verification in AutoLISP, 240
- erasing. *See* deleting
- ergonomics in dialog box design, 353
- ERRNO system variable, 486
- error codes
  - AutoLISP, 485
- Error option for protected symbols, 229
- :ERROR-BREAK trace keyword frame, 90
- errors
  - AutoLISP error handling, 254–257
  - bug introduction during optimization, 156
  - catching and continuing execution, 257–258
  - checking for syntax errors, 61–65
  - continuing after, 257–258
  - dialog box error handling
    - alert boxes, 355, 364
    - error tile guidelines, 363–364
    - forgiving errors, 355
    - for multiple DCL files, 387
    - preview errors, 346
    - semantic auditing of DCL files, 346–347, 387–388
    - testing dialog boxes with programs, 387
    - when loading first time, 386–387
  - \*error\* function, 135, 255–256
- errors (*continued*)
  - error-handling functions, 134–135, 254–256, 440
  - from ActiveX function calls, 191–193
  - handling for ActiveX methods, 191–193
  - handling for VLX application in its own namespace, 134–135
  - Inspect command error handling, 103
  - intercepting with vl-catch-all-apply, 257–258
  - MDI environment error-handling, 134–135
  - safe optimization warning messages, 158
  - separate-namespace VLX error handling, 135–136
  - viewing error trace stack, 92
  - vl-catch-all-apply, using, 257–258
  - VLX error handling, 134–135
  - See also* debugging programs
- escape codes in strings, 237–238
- establishing ActiveX connection to application, 195
- events, callback
  - defining callback functions, 205–207
  - overview, 204–205
- exclamation point (!), for viewing AutoCAD variable values, 18, 25
- executing. *See* loading; running; starting
- Existing Application Properties command (VLISP File menu), 124
- exiting
  - AutoCAD, 22
  - quietly in AutoLISP, 236
  - Visual LISP, 22
  - See also* closing
- expanding
  - element list items in Inspect window, 97
  - selections using shortcut keys, 34–35
- Expert Wizard mode, 118, 119
- exporting functions
  - to ACAD, 157
  - to document namespaces, 130–131
- Export-to-ACAD symbol flag, 95
- exposing functions. *See* exporting functions
- expressions
  - AutoLISP, 222–223
  - form for, 222
  - formatting styles for, 52
  - function syntax, 223
  - matching parentheses, 222–223
  - monitoring evaluation results of, 73–74
  - quotation marks and parentheses, 223
- EXRXSUBR Apropos filter value, 44
- EXRXSUBR Inspect window, 97, 101
- extended data, 323–330
  - associated with objects, 105, 108
  - attaching to entities, 329
  - filtering selection sets for, 300

- extended data (*continued*)
  - group codes for, 323–324
  - handles in, 330
  - managing memory use, 329
  - organization of, 324–326
  - registration of applications, 327
  - retrieving, 323, 327–328
  - synopsis of, 453
  - viewing associated with objects, 105, 108
- extensions to AutoLISP. *See* Visual LISP, extended functions

## F

- FAS Directory option (Build Options dialog box), 154

### .fas files

- described, 138–139
- directory for, 154
- loading, 138
- loading from project, 145–147

### file descriptor data type

- defined, 226
- overview, 226–227

### file extensions

- additional file types, 138–139
- .fas, 138, 139
- .lsp, 138, 139
- for project files, 139
- .vlx, 123, 125, 139

### file extensions, stripping, 234–235

### FILE Inspect window, 97, 100

### File menu (VLISP)

- Load File command, 114, 124–125
- Make Application submenu
  - Existing Application Properties command, 124
  - Make Application command, 125
  - New Application Wizard command, 117, 152
  - Rebuild Application command, 125
- New File command, 31
- Open File command, 14
- overview, 16
- Toggle Console Log command, 30
- Toggle Trace Log command, 82

### file-handling functions, 289–291, 449

- accessing help files, 290–291
- file search, 289–290
- synopsis of, 449

### files

- AutoLISP program files, 229–231
  - color coding, 231
  - comments, 230
  - creating new source files, 229
  - editing code in VLISP, 229
  - formatting code, 229–230

### files (*continued*)

- backing up automatically in text editor, 31–32
- closing in AutoLISP programs, 227
- color coding in text editor, 19, 42–43
- compiling programs from, 111–113
- Console log file, 29
- copying text to new file, 36
- creating Console log file, 30
- DCL files, 340–341
  - acad.dcl, 340
  - base.dcl, 340
  - color coding, 345
  - overview, 340–341
  - parts of, 340
  - referencing, 340
  - semantic auditing of, 346–347, 387–388
  - specifying in VLISP, 344
- FILE Inspect window, 97, 100
- file-handling functions, 289–291, 449
  - accessing help files, 290–291
  - file search, 289–290
  - synopsis of, 449
- global declarations file, 154
- input file for compiler, 112
- inserting into text editor window, 60
- LISP files to include in applications, 120
- loading, 20
- managing multiple LISP files, 138–140
- navigating in text editor, 35
- opening in text editor, 32
- output file from compiler, 113
- resource files to include in applications, 121
- restoring from backups, 32
- searching for text in, 38
- stripping file extensions, 234–235
- Support File Search Path, 112
- temporary files for projects, 154
- types, 138–140
- typographical conventions in this manual, 7
- unit definition file, 284–286
- .vlx files, 123, 125
- See also* DCL files

### fill\_image function, 383–385

### Filter Flags option (Apropos dialog box), 44

- filter lists for selection sets, 298–304
  - examples, 298–299
  - filtering for extended data, 300
  - logical grouping of filter tests, 302–303
  - overview, 298–300
  - relational tests, 300–302
  - selection set manipulation, 303–304
  - wild-card patterns in, 300
- Filter Values option (Apropos dialog box), 44

- Find command
  - Console window shortcut menu, 29
  - text editor shortcut menu, 33
  - VLISP Search menu, 37
- Find dialog box, 37–39
  - Cancel button, 37
  - Direction options, 37
  - Find button, 37
  - Find What field, 37
  - illustrated, 37
  - Mark Instances option, 38
  - Match Case option, 38
  - Match Whole Word Only option, 38
  - opening, 37
  - Search options, 37
  - See also* searching
- Find in Files option (Find dialog box), 37
- Find in Project option (Find dialog box), 37
- Find toolbar
  - Clear All Bookmarks button, 41
  - Next Bookmark button, 40, 41
  - Previous Bookmark button, 40, 41
  - Repeat Search button, 38
  - Replace button, 38
  - Search button, 37
  - Toggle Bookmark button, 40
- Find What field (Find dialog box), 37
- findfile function, 289
- finding. *See* searching
- fixed\_height (DCL attribute), 401
- fixed\_width (DCL attribute), 401
- fixed\_width\_font (DCL attribute), 401
- floating point numbers
  - REAL Inspect window, 97, 98
- floating point numbers, distance to floating point function, 280
- forcing line breaks in strings, 237–238
- foreach function, 86
- foreign-language support, 261, 359–360
- Format Code in Editor command (VLISP Tools menu), 50
- Format Date/Time option, 60
- Format Options dialog box, 51–57
  - Close Parenthesis Style options, 54
  - illustrated, 51–52
  - Insert Form-Closing Comment option, 55
  - Long List Format Style options, 56–57
  - More Options button, 51–52
  - opening, 51
  - Preserve Existing Line Breaks option, 55–56
  - Setting Case for Symbols options, 57
  - Split Comments option, 56
- formatting AutoLISP code, 50–60, 229–230
  - blocks of text, 50
  - Close Parenthesis style, 54
  - Column style, 53
  - comment styles, 58
  - formatting AutoLISP code (*continued*)
    - Format Options dialog box, 51–57
    - formatter restrictions, 59
    - in active editor window, 50–51
    - inserting form-closing comments, 55
    - long lists, 56–57
    - Narrow style, 53
    - options, 479–482
    - overriding session settings, 59
    - Plane style, 52
    - preserving existing line breaks, 55–56
    - saving formatting options, 58–59
    - selected text, 50
    - shortcut keys for, 59–60
    - Smart Indent feature, 50, 51
    - splitting long comments, 56
    - text editor feature, 19
    - Wide style, 53
- Frame Binding window, 88–92
  - adding symbol to Watch window, 89
  - calling Inspect feature, 88
  - calling Symbol Service dialog box, 89
  - copying value to \*obj\* system variable, 89
  - described, 82
  - displaying for local variables, 88
  - displaying value in Console, 88
  - keyword frames, 89–91
  - printing value to Console window, 88
  - shortcut menu commands, 88
  - special function call frames, 91
- freeing memory and releasing objects, 189
- function call frames, 86
- Function symbol command (Trace Stack window shortcut menu), 88
- :FUNCTION-ENTRY trace keyword frame, 90
- function-handling functions, 245–254
  - adding commands, 248–249
  - c:xxx functions, 247–250
  - defining functions, 245–247
  - defining functions with arguments, 252–254
  - defun function, 245–253
  - defun-q function, 251–252
  - local variables in functions, 251–252
  - redefining AutoCAD commands, 249–250
  - special forms, 253–254
  - synopsis of, 441
- functions
  - ActiveX versus ename, 162
  - application-handling, 436
  - Apropos filter value, 44
  - arithmetic, 437–438
  - as AutoCAD commands, 247–250
  - c:xxx functions, 247–250
  - call stack, 68
  - callback functions for reactors, 205–207
  - collection manipulation, 458

## functions (*continued*)

- controlling AutoCAD display, 264–268
  - graphics and text windows, 267
  - low-level graphics, 267–268
  - menus, 264–268
  - overview, 264
- conversion functions
  - angular values from radians or degrees, 281–282
  - ASCII code conversions, 282–283
  - coordinate system transformations, 286–289
  - data conversion functions, 458–459
  - measurement unit conversions, 283–286
  - point transformations, 289
  - string conversions, 279–281
  - synopsis of, 446
- curve measurement, 461–462
- Debug-on-Entry flag, 75, 94
- device access and control
  - synopsis of, 447
- device access and control functions, 291–293
  - accessing user input, 291
  - calibrating tablets, 291–293
- dialog box function synopsis, 427–429
  - application-specific data-handling functions, 429
  - dialog box opening and closing functions, 427
  - image tile-handling functions, 429
  - list box and pop-up list-handling functions, 428
  - tile- and attribute-handling functions, 428
- dialog box management functions, 365–389
  - action expressions and callbacks, 368–372
  - application-specific data, 386
  - controlling dialog boxes with AutoLISP programs, 366–368
  - DCL error handling, 386–388
  - function sequence, 388–389
  - handling tiles, 372–376
  - hiding dialog boxes, 377–379
  - image tiles and buttons, 383–385
  - list boxes and pop-up lists, 380–382
  - nesting dialog boxes, 376–377
  - sample block definition dialog box, 389
- dictionary functions, 334, 462
- display control functions, 448
- displaying definition for Symbol Service dialog box, 94
- displaying source text for, 88

## functions (*continued*)

- Drop directives and safe optimization, 158, 160
- entity access functions, 311
- entity data functions, 311–320
  - adding entities to drawings, 315–316
  - anonymous blocks, 319
  - creating complex entities, 316–318
  - deleting entities, 311
  - graphics screen and, 319–320
  - modifying entities, 314–315
  - obtaining entity information, 311–313
  - working with blocks, 319
- entity name functions, 305–311
  - entity access functions, 311
  - entity context and coordinate transform data, 307–311
  - entity handles and their uses, 307
  - overview, 305–306
  - retrieving entity names, 305–306
  - setting entity name to variable name, 306
- equality and conditional, 439
- error handling, 440
- exporting, to document namespaces, 130–132
- exposing. *See* functions, exporting
- EXRXSUBR Inspect window, 97, 101
- extended data functions, 323–330
  - attaching extended data to entities, 329
  - filtering selection sets for extended data, 300
  - group codes for extended data, 323–324
  - handles in extended data, 330
  - managing memory use, 329
  - organization of extended data, 324–326
  - registration of applications, 327
  - retrieving extended data, 323, 327–328
  - synopsis of, 453
- file-handling functions, 289–291
  - accessing help files, 290–291
  - file search, 289–290
  - synopsis of, 449
- function handling
  - synopsis of, 441
- function-handling functions, 245–254
  - adding commands, 248–249
  - c:xxx functions, 247–250
  - defining functions, 245–247
  - defining functions with arguments, 252–254
  - local variables in functions, 251–252

## functions (*continued*)

- redefining AutoCAD commands, 249–250
- special forms, 253–254
- geometric utilities, 274–279
  - finding angle between line and X axis, 274
  - finding distance between two points, 274
  - finding intersection of two lines, 274
  - finding polar coordinates of points, 274
- object snap, 274–275
- overview, 274
- synopsis of, 450
- text extents, 275–279
- help for, 50
- linking function calls, 111, 116, 159
- list handling
  - synopsis of, 442–443
- list-handling functions, 240–245
  - adding items to list beginning, 241, 241
  - combining lists, 241
  - dotted pairs, 244–245
  - grouping related items, 240
  - point lists, 242–244
  - retrieving items from lists, 240–241
  - returning all but first element, 241
  - substituting items, 241, 241
- as lists, 247–250
- local variables in, 251–252
- Make Application output, 123
- making available
  - as AutoCAD commands, 247–250
  - to documents, 130–132
  - to ObjectARX applications, 95
  - to other VLX applications, 132
- memory management functions, 457
- method invocation functions, 459
- multiple namespaces affect on, 128
- namespace communication functions, 468
- number handling in AutoLISP, 233
- object-handling functions
  - entity access functions, 311
  - entity data functions, 311–320
  - entity name functions, 305–311
  - non-graphic object-handling, 321–323
  - polylines, old-style and lightweight, 320–321
  - synopsis of, 453–454, 463
- output functions, 235–239
  - control characters in quoted strings, 237–238
  - displaying messages, 236

## functions (*continued*)

- wild-card matching, 238–239
- output windows for, 15
- property-handling functions, 460
- protected symbols and, 94
- query and command functions
  - configuration control, 264
  - foreign-language support, 261
  - inspecting and changing system and environment variables, 263–264
  - passing pick points to AutoCAD commands, 262–263
  - pausing for user input, 261–262
  - sending commands to AutoCAD prompt, 260–261
  - synopsis of, 451
  - undoing commands issued with command function, 263
- querying reactors using function calls, 213
- reactor callback functions, 205–207
- reactor functions, 464–466
- restrictions when dialog box is open, 367–368
- running from Console prompt, 20–21
- selection set-handling functions
  - adding entities, 303–304
  - creating selection sets, 296–297
  - deleting entities, 303–304
  - finding number of entities, 304
  - passing selection sets between AutoLISP and ObjectARX applications, 305
  - returning entity names, 304
  - selection set filter lists, 298–304
  - synopsis of, 455
  - testing whether an entity is a member, 304
- special forms, 253–254
- string handling, 444–445
- SUBR Inspect window, 97, 100
- summary of
  - basic functions, 436–445
  - category summary, 434–436
  - memory management functions, 457
  - namespace communication functions, 468
  - reactor functions, 464–466
  - selection set, object, and symbol table functions, 453–457
  - symbol-handling functions, 445
  - utility functions, 446–452
  - Visual LISP extended functions, 457–460
  - VLX namespace functions, 467
  - Windows Registry functions, 469
- symbol table access functions, 332–333

## functions (*continued*)

- syntax conventions in AutoLISP, 223
- trace stack and, 86
- tracing user-defined functions, 94
- typographical conventions in this manual, 7
- user input functions
  - accessing user input from devices, 291
  - allowable input, 268–269
  - arbitrary keyboard input, 273
  - controlling user-input conditions, 271–273
  - getting user input, 268–271
  - getxxx functions, 268–271
  - input options, 271–272
  - keyword options, 272–273
  - pausing for user input, 261–262
  - restrictions when dialog box is open, 367
  - synopsis of, 452
  - validating input, 273
- user-defined
  - defining with defun, 245–253
  - protected symbols and, 94
  - tracing, 94
- using with ActiveX methods, 170–191
  - applying object's property to new object, 181
  - AutoLISP data types accepted in place of ActiveX data type, 180
  - changing X axis of circle, 181–183
  - converting AutoLISP data types to ActiveX data types, 173–180
  - converting object references, 190–191
  - determining function needed, 171–172
  - determining how to call functions, 172–173
  - determining if method or property applies to object, 186–187
  - determining if object is available for updating, 183
  - listing object properties and methods, 185–187
  - reading object properties, 180–181
  - releasing objects and freeing memory, 189
  - updating object properties, 181–183
  - using ActiveX methods that return values in arguments, 184–185
  - vla- functions, 170
  - vla-get functions, 170
  - vla-put functions, 170
  - vla-x- functions, 170
  - working with collection objects, 187–189

## functions (*continued*)

- USUBR Inspect window, 97, 100
- Visual LISP extensions, 22, 457–463
  - collection manipulation functions, 458
  - curve measurement functions, 461–462
  - data conversion functions, 458–459
  - defined, 22
  - dictionary functions, 462
  - loading extended functions, 22
  - method invocation functions, 459
  - object-handling functions, 463
  - property-handling functions, 460
  - running, 22
  - vl-load-com and, 22
  - VLX namespace functions, 467
  - VLX namespace functions, 467
  - Windows Registry functions, 469
- functions (Apropos filter value), 44
- fundamental units, 284

## G

- geometric utilities, 274–279, 450
  - finding
    - angle between line and X axis, 274
    - distance between two points, 274
    - intersection of two lines, 274
    - polar coordinates of points, 274
  - object snap, 274–275
  - overview, 274
  - synopsis of, 450
  - text extents, 275–279
- get\_attr function, 374
- getangle function, 269–270
- GetBoundingBox method, 184
- getcorner function, 268–269
- getdist function, 269
- getenv function, 264
- getfiled function, 289–290
- getint function, 268–269
- getkeyword function, 269, 270
- getorient function, 269–270
- getpoint function, 268–269
- getreal function, 268, 269
- getstring function, 268, 269
- getting user input, 268–273
  - accessing user input from devices, 291
  - allowable input, 268–269
  - arbitrary keyboard input, 273
  - controlling user-input function conditions, 271–273
  - getxxx functions, 268–271
  - input options, 271–272
  - keyword options, 272–273
  - pausing for user input, 261–262
  - validating input, 273

- getvar function, 263
- getxxx functions, 268–273
- Global Declarations file for projects, 154
- Go to Last Edited command (text editor shortcut menu), 33
- graphics screen and entity data functions, 319–320
- graphscr function, 267
- grdraw function, 267–268
- group codes for regular and extended data, 323–324
- gread function, 291
- grtext function, 267–268
- grvecs function, 267–268

## H

- handent function, 307, 330
- handles
  - entity handles and their uses, 307
  - in extended data, 330
  - searching for ename associated with, 190
  - searching for object associated with, 191
  - searching for, associated with ename, 190
  - searching for, associated with object, 191
- height (DCL attribute), 401–402
- Help command (Apropos results window shortcut menu), 47
- help feature
  - accessing Help files, 290–291
  - for ActiveX object methods, 164
  - described, 3
  - for Apropos results, 46
  - for AutoLISP functions, 50
  - providing dialog box help, 355
  - Symbol Service dialog box, 94
- help function, 290
- Help menu (VLISP), 17
- .hh files, 140
- hiding dialog boxes
  - canceling and, 377
  - design guidelines, 358
  - handling, 377–379
- highlight colors for breakpoints, 79
- history of Console input
  - retrieving previous commands, 18, 26–27
  - searching, 18, 27
- .hpp files, 140

## I

- IDE. *See* integrated development environment (IDE)
- image tiles. *See* tiles (dialog box)
- image\_button tile, 416–417
- importing type libraries, 193–195
- improper lists, 244
- inches, converting to meters, 284

- Indent Block option, 59
- Indent to Current Level option, 59
- indenting code
  - shortcut keys for, 59
  - using text editor shortcut keys, 36
- initget function, 270, 271–273
- initial\_focus (DCL attribute), 402
- input history. *See* history of Console input
- input, user. *See* dialog box design, user input; getting user input
- Inquire Extended Data command
  - Inspect window shortcut menu, 105
  - VLISP View menu, 107
- Insert Date option, 60
- Insert File option, 60
- Insert Form-Closing Comment option (Format Options dialog box), 55
- Insert Time option, 60
- inserting
  - files into text editor window, 60
  - form-closing comments, 55
  - shortcut keys for, 60
  - symbols from Apropos results window, 46
  - using text editor overstrike mode, 34
- Inspect command
  - Apropos results window shortcut menu, 46
  - Console window shortcut menu, 29
  - error handling, 103
  - Frame Binding window shortcut menu, 88
  - Inspect window shortcut menu, 102
  - text editor shortcut menu, 33
  - Trace Stack window shortcut menu, 88
- Inspect Drawing Objects Verbosely option, 104
- Inspect feature
  - caption (title bar) of window, 96
  - closing all Inspect windows, 103
  - copying Inspect window contents to Trace window and log, 102
  - copying objects to \*obj\* system variable, 103
  - described, 3, 68, 82
  - element list content, 96
  - element list formats, 97–102
  - error handling, 103
  - expanding element list items, 97
  - Inspect window contents, 96–97
  - inspecting reactors, 212
  - invoking
    - for symbol in Apropos results window, 46
    - from Frame Binding window, 88
    - from Symbol Service dialog box, 94
    - from Trace Stack window, 88
    - from Watch window, 84
    - Symbol Service feature, 102
  - maximum element lines per window, 96
  - navigating multiple pages, 96

Inspect feature (*continued*)

- object line of window, 96
- opening Inspect window, 29, 33, 95
- opening text editor window containing selected text, 103
- overview, 95–96
- passing element values as argument, 102
- printing Inspect window object to Console window, 102
- shortcut menu commands, 102, 105
- updating Inspect window, 102
- viewing AutoCAD drawing entities, 104–108
  - blocks in the drawing database, 106–107
  - controlling amount of Inspect information, 104
  - entities in the drawing database, 104–105
  - extended data associated with objects, 105, 107–108
  - selected objects in drawings, 107
  - symbol tables in the drawing database, 105
- viewing AutoCAD object properties, 166–169

Inspect Next Entity command (Inspect window shortcut menu), 105

Inspect Raw Data command (Inspect window shortcut menu), 105

Inspect Value command (Watch window shortcut menu), 84

Inspect window. *See* Inspect feature

:INSPECT-EVAL trace keyword frame, 89

inspecting. *See* displaying; Inspect feature

inspection tools. *See* Frame Binding window; Inspect feature; Symbol Service dialog box; Trace Stack window

:INSPECT-VERBOSE trace keyword frame, 89

INT Inspect window, 98

integer data type
 

- for DCL attribute values, 392
- defined, 224
- number handling in AutoLISP, 233
- overflow handling by AutoLISP, 224–225
- overview, 224–225

integrated development environment (IDE)
 

- components, 2–3
- returning to, 12
- starting Visual LISP, 12, 22
- Visual LISP as, 1

interactive programming in Console window, 25

intercepting program errors, 257–258

international language considerations, 261, 359–360

interrupting commands, 27

is\_bold (DCL attribute), 402

is\_cancel (DCL attribute), 402

is\_default (DCL attribute), 402–403

is\_enabled (DCL attribute), 403

is\_tab\_stop (DCL attribute), 403

italic type in this manual, 7

## K

:KBD-BREAK trace keyword frame, 90

key (DCL attribute), 403

keyboard input and dialog box design, 359

keyboard shortcuts. *See* shortcut keys

keyword frames (in trace stack)
 

- overview, 86, 89
- types at bottom of stack, 89
- types at top of stack, 90–91

## L

label (DCL attribute), 404

lambda forms, 86

lambda function, 86

language in dialog box design
 

- avoiding abbreviations, 357
- capitalization, 356–357
- clarity, 353
- international language considerations, 359–360

languages, supporting foreign, 261, 359–360

layout (DCL attribute), 404

layout of dialog boxes, 347–352
 

- adjusting space at right side or bottom, 350
- adjusting space between tiles, 349
- centering buttons, 347–348
- customizing exit button text, 351–352
- distributing tiles in a cluster, 348–349
- example, 347–348
- fixing spacing around boxed row or column, 350–351

life cycle of breakpoints, 81

lines (graphic)
 

- finding angle between line and X axis, 274
- finding intersection of two lines, 274
- old-style and lightweight polylines, 320–321
- processing curve-fit and spline-fit polylines, 321

lines (text)
 

- deleting from cursor to end of line, 60
- forcing line breaks in strings, 237–238
- navigating in text editor, 35
- Preserve Existing Line Breaks option, 55–56
- running selected lines of code, 21
- starting new lines in text editor, 31

Link mode option (Build Options dialog box), 155

Linker reactors, 203



- linking
    - choosing a link mode, 155, 157
    - function calls, 111, 116
    - safe optimization and, 158, 159
    - See also* compiling and linking programs
  - LISP Files to Include dialog box, 119, 152
  - LISP, reasons for choosing, 2
  - list (DCL attribute), 404
  - list boxes. *See* tiles (dialog box)
  - list data type
    - defined, 225
    - overview, 225
  - list function
    - forming point lists, 242
    - grouping related items, 240
  - list handling
    - adding items to list beginning, 241
    - association lists, 244
    - combining lists, 241
    - creating lists, 240–245
    - dotted pairs, 244–245
    - grouping related items, 240
    - improper lists, 244
      - viewing, 97, 99
    - point lists, 242–244
    - proper lists, 244
      - viewing, 97, 99
    - retrieving items from lists, 240–241
    - returning all but first element, 241
    - substituting items, 241
    - synopsis of, 442–443
  - LIST Inspect window, 97, 99
  - list\_box tile, 417
  - listing
    - breakpoints, 80
    - condition-function pairs for reactors, 213
    - functions exported by applications, 132
    - object properties and methods, 185–187
    - objects that fire notifications to reactors, 213
    - reactor callback events, 205
    - reactor types, 204
    - reactors of given type, 211–212
    - separate-namespace applications associated with documents, 132
  - lists, AutoLISP. *See* list handling
  - Load Application command (VLISP Tools menu), 124
  - Load command (Project window shortcut menu), 146
  - Load File command (VLISP File menu), 114, 124
  - load function
    - loading compiled programs, 114
    - loading VLX applications, 124
  - Load Project FAS button (Project window), 145, 147
  - Load Source command (Project window shortcut menu), 146
  - Load Source Files button (Project window), 147
  - Load Text command (VLISP Tools menu), 20
  - load\_dialog function, 366, 387
  - loading
    - ActiveX support, 165
    - application in its own namespace, 128–129
    - AutoLISP programs from text editor, 20
    - compiled programs, 114
    - dialog boxes, 366, 387
    - order for application file loading, 120
    - order for project file loading, 144
    - project .fas files, 146–148
    - project source files, 145–147
    - reactor support, 202, 205
    - selected lines of code, 21
    - VLX applications, 124
    - See also* opening; running; starting
  - loading compiled programs, 114
  - loading VLX applications, 124
  - local variables, 251–252
  - Local Variables command (Trace Stack window shortcut menu), 88
  - Localize directives and safe optimization, 158, 160
  - Localize Variables option (Build Options dialog box), 155
  - Log command (Inspect window shortcut menu), 102
  - logging Console window activity, 29–30
    - appending information to existing file, 30
    - creating log files, 30
    - quitting the logging process, 30
    - replacing existing file, 30
  - logging Trace activity
    - copying Apropos search results to log file, 46
    - turning on and off, 82
  - Long List Format Style options (Format Options dialog box), 56–57
  - long lists
    - defined, 56
    - format style options, 56–57
  - lsa compiler mode parameter, 111
  - lsm compiler mode parameter, 111
  - .lsp files
    - described, 138–139
    - finding strings in, 151
    - loading from project, 145–146
- ## M
- Make Application command (VLISP File menu), 125
  - Make Application Wizard, 117–123, 152
    - application directory, 118
    - application options, 118

- Make Application Wizard (*continued*)
  - building the application, 123
  - compilation options, 122–123
  - including projects in applications, 152
  - LISP files to include, 119, 152
  - naming applications, 118
  - order for loading application files, 120
  - resource files to include, 121, 152
  - Wizard mode, 117
- making application modules
  - changing application options, 124–125
  - creating new applications, 117
  - including projects, 151
  - loading and running VLISP applications, 124
- Make Application output, 123
- object directory for, 125
- rebuilding applications, 125
- updating applications, 125
- using Make Application Wizard. *See* Make Application Wizard
- managing dialog boxes. *See* dialog box management
- managing VLISP applications, 137–160
  - defining projects, 140–149
    - assigning project properties, 141–144
    - overview, 140
    - using the Project window, 145–149
  - managing multiple LISP files, 138–140
  - optimizing application code, 153–160
    - choosing a Compilation mode, 155–156
    - choosing a Link mode, 155, 157
    - defining build options, 153–155
    - safe optimization, 157–160
  - working with existing projects, 149–152
    - finding strings in project source files, 151
    - including projects in VLISP applications, 151
    - opening projects, 149
- manipulating AutoCAD objects, 295–334
  - extended data, 323–330
  - object-handling, 305–323
  - selection set handling, 296–305
  - symbol table and dictionary access, 332–334
  - xrecord objects, 331
- mapchar function, 381
- Match Backward command (VLISP Edit menu), 61–63
- Match by Prefix option (Apropos dialog box), 43
- Match Case option (Find dialog box), 38
- Match Forward command (VLISP Edit menu), 61–63
- Match Whole Word Only option (Find dialog box), 38
- matching parentheses
  - in AutoLISP code, 19
- matching parentheses in AutoLISP code, 61–63, 222–223
- matching wild-cards in strings, 238–239
- max\_value (DCL attribute), 405
- MDI (multiple document interface), and
  - AutoLISP, 126–134
  - AutoLISP limitations, 136
  - error-handling in MDI environment, 134–135
  - limitations, AutoLISP, 136
  - namespaces overview, 126–128
  - running an application in its own namespace, 129–130
  - sharing data between namespaces, 133–134
- measurement unit conversions, 283–286
- memory
  - freeing, 305, 457
  - garbage collection, 305, 457
  - managing extended data memory use, 329
  - memory management functions, 457
  - releasing objects and freeing memory, 189
- menucmd function, 264–268
- menus (VLISP)
  - See also* specific menus
- menus (AutoCAD)
  - controlling, 264–268
  - customizing. *See* AutoCAD Customization Guide
- menus (VLISP)
  - issuing VLISP commands from, 15
  - overview, 13, 15–17
  - shortcut menus
    - Apropos results window, 46–47
    - Console window, 29
    - displaying, 29
    - Frame Binding window, 88
    - Inspect window, 102
    - Project window, 146
    - text editor, 32–33
    - Trace Stack window, 87–88
    - Watch window, 84
    - variable contents of, 15–16
- Merge Files mode option (Build Options dialog box), 154
- Message mode option (Build Options dialog box), 155
- messages, safe optimization warning, 158
- meters, converting inches to, 284
- method invocation functions, 459
- methods, ActiveX. *See* ActiveX, methods
- min\_value (DCL attribute), 405
- mnemonic (DCL attribute), 405
- mode codes for, 373
- mode\_tile function, 372–374

- Modify command (Inspect window shortcut menu), 105
- monitoring
  - evaluation results of an expression, 73–74
  - variables during program execution, 73–74
- monospace font conventions in this manual, 7
- mouse
  - Console window shortcut menus, 29
  - displaying shortcut menus, 29
  - dragging text in text editor, 36
  - See also* dialog box design, user input; getting user input
- moving around. *See* navigating
- moving text
  - by dragging in text editor, 36
  - Clipboard support for, 36
  - undoing move actions, 37
  - using Console window shortcut menu, 29
  - using text editor shortcut menu, 32–33
- multiple document environment
  - designing for. *See* designing for multiple document environments
- multiple documents. *See* MDI
- multiple LISP files, managing, 138–140
- multiple namespaces. *See* namespaces
- Multiple Selection command (Project window shortcut menu), 146
- multiple\_select (DCL attribute), 406

## N

- names/naming
  - DCL attributes, 367–369
  - entity name functions, 305–311
  - input file for compiler, 111–112
  - Make Application output, 123
  - output file from compiler, 113
  - setting entity name to variable name, 306
  - stripping file extensions, 234–235
  - symbol naming restrictions, 227
  - symbol table entries that cannot be renamed, 322–323
  - tiles (dialog box), 342, 343
  - typographical conventions in this manual, 7
  - variables, 228
- namespaces
  - blackboard, 133
  - defined, 126
  - error-handling in MDI environment, 134–135
  - functions and multiple, 128–129
  - making functions available to documents, 130–132
  - making functions available to other VLX applications, 132
  - namespace communication functions, 468
  - overview, 126

- namespaces (*continued*)
  - referencing variables in document namespaces, 132–133
  - running an application in its own namespace, 129–130
  - sharing data between, 133–134
  - variables and multiple, 126–128
  - VLX namespace functions, 467
- Narrow formatting style, 53
- navigating
  - going to position last edited, 35
  - Inspect window pages, 96
  - moving insertion point to parentheses, 61–63
  - using bookmarks, 40–41
  - using text editor shortcut keys, 35
- nentsel function, 305–306, 307–311
- nentselp function, 307–308, 310
- nesting dialog boxes
  - canceling all dialog boxes, 368, 377, 378
  - design guidelines, 358
  - managing, 376–377
- New Application Wizard command (VLISP File menu), 110, 117–122, 131, 152
- New File command (VLISP File menu), 31, 229
- new\_dialog function
  - default actions, 369, 372
  - error handling, 387
  - nested dialog boxes and, 376, 377
  - nil values and, 367
  - passing dialog name and ID, 366
  - Windows and, 367
- newline character, 237
- Next Bookmark command (VLISP Search menu), 40, 41
- nil variables
  - exiting quietly, 236
  - overview, 232
- non-continuable break loops, 78
- Nonnull (Apropos filter value), 44
- notifications, reactor, 204–205
- nth function, 240–241
- ntmod function, anonymous blocks and, 319
- Null (Apropos filter value), 44
- number handling in AutoLISP, 233

## O

- .ob files, 139
- Object Coordinate System, 287
- object directory, for applications, 125
- Object ID, ARX, 190, 191
- object model, AutoCAD. *See* AutoCAD object model
- Object reactors
  - attaching data to reactor objects, 210
  - defining callback functions, 207
  - overview, 203

## Object reactors (*continued*)

- seeing how they work, 209–210
- using, 208–210

## Object Snap modes, 274–275

## ObjectARX applications

- called from AutoLISP programs, 156
- making functions available to, 95
- passing selection sets between AutoLISP and, 305

## ObjectARX Reference, 8

## objects

- accessing in ActiveX, 165–169
  - circle object example, 168–169
  - moving forward from the application object, 168
- overview, 165, 168–169
- returning pointer to AutoCAD object, 165
- specifying VLA-object type, 165
- using Inspect tool to view object properties, 166–169
- ActiveX objects, 162
- applying properties to new object, 181
- ARX Object ID of, 190, 191
- attaching data to reactor objects, 210
- changing. *See* objects, modifying
- collection objects
  - collection manipulation functions, 458
  - evaluating series of functions with each object in, 187–188
  - overview, 164
  - retrieving member objects, 188–189
  - working with, 187–189
- converting between enames and VLA-objects, 190
- converting object references, 190–191
- creating new instance of application object, 195
- determining availability for updating, 183
- determining if method or property applies, 186–187
- dictionary objects, 323
- entity access functions, 311
- entity data functions, 311–320
  - adding entities to drawings, 315–316
  - anonymous blocks, 319
  - creating complex entities, 316–318
  - deleting entities, 311
  - graphics screen and, 319–320
  - modifying entities, 314–315
  - obtaining entity information, 311–313
  - working with blocks, 319
- entity name functions, 305–311
  - entity access functions, 311

## objects (*continued*)

- entity context and coordinate transform data, 307–311
- entity handles and their uses, 307
- overview, 305–306
- retrieving entity names, 305–306
- setting entity name to variable name, 306
- identifying ARX Object ID of, 191
- listing properties and methods, 185–187
- manipulating AutoCAD objects, 295–334
  - extended data, 323–330
  - object-handling, 305–323
  - selection set handling, 296–305
  - symbol table and dictionary access, 332–334
  - xrecord objects, 331
- methods, 164
- modifying
  - with ActiveX, 180–183
  - collection objects, 187
  - with entmod, 314–315
- object-handling functions, 305–323
  - entity access functions, 311
  - entity data functions, 311–320
  - entity name functions, 305–311
  - non-graphic object-handling, 321–323
  - polylines, old-style and lightweight, 320–321
  - synopsis of, 453–454, 463
- obtaining one object identifier from another, 190–191
- PICKSET Inspect window, 98, 101
- properties of, in ActiveX and VBA
  - Reference, 164
  - changing with ActiveX, 181–183
  - defined, 164
  - getting, 180–181
  - listing, 185–187
  - modifying with ActiveX, 181–183
  - reading, 180–181
  - updating with ActiveX, 181–183
  - viewing, 180–181
- reading object properties, 180–181
- releasing objects and freeing memory, 189
- searching for handle associated with, 191
- searching for object associated with handle, 191
- selection set handling functions
  - adding entities, 303–304
  - creating selection sets, 296–297
  - deleting entities, 303, 304
  - finding number of entities, 304
  - passing selection sets between AutoLISP and ObjectARX applications, 305

## objects (*continued*)

- returning entity names, 304
- selection set filter lists, 298–304
- synopsis of, 455
- testing whether an entity is a member, 304
- symbol table objects, 322–323
- testing, availability for modification, 183
  - availability for reading, 183
  - availability for updating, 183
  - whether erased or not, 183, 183
  - whether method applies to, 186
  - whether property applies to, 186–187
  - whether released or not, 189
- translating from ARX Object ID to object, 191
- updating object properties, 181–183
- VLA-object type specification, 165
- xrecord objects, 331
- obtaining entity information, 311–313
- obtaining user input. *See* getting user input
- ok\_cancel tile, 343, 418
- ok\_cancel\_help tile, 418
- ok\_cancel\_help\_errtile tile, 418–419
- ok\_cancel\_help\_info tile, 419
- ok\_only tile, 418
- Open File command (VLISP File menu), 14
- Open Project command (VLISP Project menu), 149, 153
- opening
  - Apropos window, 29, 33
  - AutoLISP source files in text editor, 14
  - dialog box opening functions, 427
    - drawings containing persistent reactors, 215–216
  - files in text editor, 32
  - Format Options dialog box, 51
  - Inspect window, 29, 33, 94, 95
  - new files in text editor, 31
  - Project Properties dialog box, 142, 145
  - projects, 149
  - selecting text before opening files, 32
  - source files in text editor, 14
  - starting Visual LISP, 12, 22
  - Symbol Service dialog box, 29, 33, 46, 89, 93, 102
  - Watch window, 29, 33
- operators, relational, for selection set filter lists, 300–302
- optimizing application code, 153–160
  - analyzing for optimization correctness, 156
  - bug introduction during, 156
  - choosing a compilation mode, 111, 154, 155
  - choosing a Link mode, 155, 157
  - defining build options, 153–155
  - safe optimization, 157–160

## optimizing application code (*continued*)

- See also* safe optimization
- order for loading
  - application files, 120
  - program files, 114
  - project AutoLISP files, 144
- osnap function, 274–275
- output functions, 235–239
  - control characters in quoted strings, 237–238
  - displaying messages, 236
  - wild-card matching, 238–239
- overriding format settings, 59
- overstrike mode in text editor, 34

## P

- Pa symbol flag, 94
- Paper Space DCS, 287
- paragraph tile, 419–420
- parentheses
  - checking balance, 19, 61–63
  - Close Parenthesis Style options, 54
  - matching in AutoLISP code, 19, 61–63, 222–223
  - moving insertion point to, 61–63
  - navigating in text editor, 35
  - selecting text between cursor and, 62
  - selecting text in text editor and, 34
  - Smart Indent feature and, 51
  - unbalanced, found by formatter, 50
- passing pick points to AutoCAD commands, 262–263
- passing selection sets between AutoLISP and ObjectARX applications, 305
- password\_char (DCL attribute), 379, 406
- passwords, requesting from users, 379
- Paste command
  - Console window shortcut menu, 29
  - text editor shortcut menu, 33
- pasting. *See* copying; moving text
- paths
  - Make Application output, 123
  - for project files, 143, 144
  - for source files, 111–112
  - Support File Search Path, 112
- PAUSE symbol, 232, 261–262
- pausing for user input, 261–262
- .pdb files, 139
- persistent reactors
  - defined, 215
  - making, 215
  - opening drawings containing, 215–216
  - transient reactors versus, 215
- PI variable, 232
- pick points, passing to AutoCAD commands, 262–263
- PICKSET Inspect window, 98, 101

- placement of dialog boxes, 357
- Plane style, 52
- point lists, 242–244
- points
  - coordinate system transformations, 286–289
  - finding distance between, 274
  - finding polar coordinates of, 274
  - transformations, 289
- polylines
  - old-style and lightweight, 320–321
  - processing curve-fit and spline-fit polylines, 321
- populating safearrays with data, 175–178
- popup\_list tile, 420–421
- predefined attributes, synopsis of, 394–396
- predefined tiles and clusters (dialog box design guidelines), 360–364
  - buttons, 360
  - clusters, 361
  - edit boxes, 361
  - image buttons and image tiles, 361–362
  - list boxes, 362
  - radio buttons, radio rows and radio columns, 360–362
  - sliders, 362
  - text, 363
  - toggles, 363
- predefined variables, AutoLISP, 232–233
- Prefix With option, 59
- Preserve Existing Line Breaks option (Format Options dialog box), 55–56
- Pretty Print command (Inspect window shortcut menu), 102
- Preview DCL in Editor command (VLISP Tools menu), 345
- Previous Bookmark command (VLISP Search menu), 41
- prin1 function, 236
- princ function
  - exiting quietly, 236
  - output display from, 18, 236
- Print command
  - Apropos results window shortcut menu, 46
  - Frame Binding window shortcut menu, 88
  - Inspect window shortcut menu, 102
  - Trace Stack window shortcut menu, 88
- print function, 18, 236
- Print message option for protected symbols, 228
- Print Value command (Watch window shortcut menu), 85
- printing
  - files, 16
  - Frame Binding window value to Console window, 88
  - Inspect window object to Console window, 102
- printing (*continued*)
  - list of arguments to Trace window, 206
  - messages in AutoLISP, 236
  - stack element to Console window, 88
  - symbol name from Apropos in Console window, 46
  - Watch window variable values in Console window, 85
- .pj files
  - closing, 149
  - described, 139
- progn expressions, Narrow style for, 53
- programs. *See* AutoLISP programs
- Project menu (VLISP)
  - Open Project command, 149, 153
  - overview, 17
- Project Properties button (Project window), 145, 153
- Project Properties dialog box
  - changing order for loading files, 144
  - compiler build options, 144, 153–155
  - identifying path name of files, 143–144
  - opening, 148, 153
  - selecting files to include in project, 141–143
- Project window
  - adding files to project, 146
  - arranging files, 146
  - buttons, 145
  - checking syntax of source code, 146
  - closing projects, 149
  - compiling project files, 148
  - editing source code, 146, 148
  - loading compiled files, 146–148
  - loading source files, 146
  - recompiling project files, 148
  - removing members, 146
  - saving projects, 146, 149
  - selecting multiple members, 146–147
  - shortcut menu commands, 145
- projects
  - adding files, 146
  - checking syntax of source code, 146
  - closing, 149
  - compiler build options, 144, 153–155
  - compiling source files, 145, 148
  - defining, 140–149
    - assigning project properties, 141–144
    - overview, 140
    - using the Project window, 145–149
  - editing source code, 146, 148
  - file types, 138–140
  - finding strings in project source files, 151
  - global declarations file, 154
  - home directory, 141
  - including in VLISP applications, 151
  - loading compiled files, 145–148

## projects (*continued*)

- loading source files, 145, 146
  - opening, 149
  - order for loading files, 144
  - overview, 138
  - path name of files, 143–144
  - recompiling source files, 145, 148
  - removing members, 146
  - saving, 146, 149
  - searching for text in, 38
  - selecting files to include, 141–143
  - selecting multiple members, 146–147
  - source directory, 141
- prompt function, 236
- Prompt to enter break loop option for protected symbols, 228
- proper lists, 244
- properties of objects. *See* objects, properties
- property-handling functions, 460
- Protect-Assign symbol flag, 94
- :PROTECT-ASSIGN trace keyword frame, 90
- protected symbols, 228–229
- prototypes. *See* tiles (dialog box)

## Q

- query and command functions
- configuration control, 264
  - foreign-language support, 261
  - inspecting and changing system and environment variables, 263–264
  - passing pick points to AutoCAD
    - commands, 262–263
  - pausing for user input, 261–262
  - sending commands to AutoCAD prompt,
    - 260–261
  - synopsis of, 451
  - undoing commands issued with command function, 263
- querying reactors
- listing all reactors of given type, 211–212
  - using function calls, 213
  - using Inspect tool, 212
- Quit Current Level command (VLISP Debug menu), 77
- quitting. *See* exiting
- quotation marks (")
- and parentheses in expressions, 223
  - using within quoted strings, 237
- quote function, forming point lists, 242
- quoted lists
- Column style for, 53
  - Long List Format Style options, 56–57
- quoted strings
- control characters in, 237–238, 392
  - for DCL attribute values, 392

## R

- radians
- converting degrees to, 281–282
  - converting to degrees, 281–282
- radio buttons, radio rows, and radio columns. *See* tiles (dialog box)
- radio\_button tile, 421
- radio\_column tile, 422
- radio\_row tile, 422
- random access memory (RAM). *See* memory
- :REACTOR-CALLBACK trace keyword frame, 90
- reactors
- adding owner object, 214
  - attaching data to reactor objects, 210
  - beeping your PC, 206
  - callback events, 204–205
  - callback functions, defining, 205–207
  - changing application-specific data
    - associated with, 213
  - changing callback function link, 213
  - communicating with applications, 202
  - creating, 207–210
  - defined, 202
  - defining callback functions, 205–207
  - determining if active, 214
  - determining if persistent or transient, 215
  - disabling, 214–215
  - enabling, 215
  - events, 204–205
  - function reactors, summary, 464–466
  - function synopsis, 464–466
  - guidelines for using, 216–217
  - inspecting, 212
  - listing
    - all reactors of given type, 211–212
    - callback events, 205
    - condition-function pairs, 213
    - objects that fire notifications, 213
    - reactor types, 204
  - loading support for, 202, 205
  - making persistent, 215
  - modifying definitions, 213–214
  - notifications, 204–205
  - opening drawings containing persistent reactors, 215–216
  - persistence, 215–216
  - printing list of arguments to Trace window, 206
- querying
- for application-specific data value, 213
  - listing all reactors of given type,
    - 211–212
  - listing condition-function pairs, 213
  - listing objects that fire notifications,
    - 213
  - for name of event triggering callback function, 213

- reactors (*continued*)
  - for reactor type, 213
  - using function calls, 213
  - using Inspect tool, 212
- removing, 214–215
- removing owner object, 214
- transient versus persistent, 215
- types, 202–204
  - Database reactors, 202
  - Document reactors, 202
  - Editor reactors, 203–204
  - Linker reactors, 203
  - listing all types, 204
  - Object reactors, 203
  - using Object reactors, 208–210
- read function, 382
- read-char function, 291
- :READ-ERROR trace keyword frame, 91
- reading object properties, 180–181
- read-line function, 291
- real data type
  - converting to string, 279–280
  - for DCL attribute values, 392
  - defined, 225
  - number handling in AutoLISP, 233
  - overview, 225
  - scientific notation for, 225
- REAL Inspect window, 97, 98
- Rebuild Application command (VLISP File menu), 125
- Rebuild Project FAS button (Project window), 145
- rebuilding applications, 125
- recompiling project source files, 145, 148
- redefining AutoCAD commands, 249–250
- Redo command
  - Console window shortcut menu, 29
  - text editor shortcut menu, 33
  - VLISP Edit menu, 31
- redraw function, 267
- referencing
  - DCL files, 340
  - tiles (dialog box), 343
  - variables in document namespaces, 132–133
- refreshing Trace Stack window contents, 92
- regapp function, 327–328
- registration of applications, 327
- Registry functions for Windows, 469
- relational operators for selection set filter lists
  - logical grouping of, 302–303
  - overview, 300–302
- releasing objects and freeing memory, 189
- Remove File command (Project window shortcut menu), 146
- Remove from Watch command (Watch window shortcut menu), 85
- removing. *See* deleting
- repeat function, 86
- repeating earlier searches, 39
- Replace command (VLISP Search menu), 39
- Replace dialog box, 39
- replacing
  - Console log file, 30
  - list items, 241, 241
  - text using overstrike mode, 34
  - text using search-and-replace, 39
- requesting passwords from users, 379
- reserved words for DCL attribute values, 392
- Reset to Top Level command (VLISP Debug menu), 77, 92
- Resource Files to Include dialog box, 121, 152
- restoring from backup files in text editor, 32
- restrictions
  - formatter, 59
  - functions restricted when dialog box is open, 367–368
  - restricted characters, 227
  - restricted DCL attributes, 393
  - symbol naming restrictions, 227
- retrieving
  - commands in Console window
    - searching for specific commands, 18, 27
    - using TAB key, 18, 26–27
  - entity names, 305–306
  - extended data, 105, 108, 323, 327–328
  - items from lists, 240–241
  - member objects in collections, 188–189
  - variable values from document namespace, 132–133
- return character in quoted strings, 238
- returning to the VLISP IDE, 12
- row tile, 423
- rtos function, 279–280
- Run toolbar
  - Activate AutoCAD button, 21
  - Load Active Edit Window button, 20
  - Load Selection button, 21
- running
  - application in its own namespace, 129–130
  - applications, 114–115, 124
  - AutoCAD with Visual LISP, 3, 20
  - AutoLISP programs from Console prompt, 20–21
  - Check command, 65
  - compiled programs, 114–115
  - in Animate mode, 74
  - selected lines of code, 21
  - separate-namespace applications, 129–132
  - starting Visual LISP, 12, 22
  - See also* loading; opening; starting



## S

- safe optimization, 157–160
  - build option, 155
  - compiler checking of conditions, 159
  - conditions bypassed by, 158
  - overview, 157
  - warning messages, 158
- Safe Optimize option (Build Options dialog box), 155
- SAFEARRAY Inspect window, 98, 102
- safearrays
  - converting to lists, 177
  - creating, 175–177
  - creating variant with an array of four doubles, 179
  - displaying contents of, 177
  - populating with data, 175–177
  - using with variants, 178–179
  - working with, 175–178
- sample files, AutoLISP, 14
- Save Block As option, 36, 60
- Save Project As command (Project window shortcut menu), 146
- Save Settings command (VLISP Tools menu), 58
- saving
  - changes to code when exiting AutoCAD, 22
  - formatting options, 58–59
  - projects, 146, 149
- scientific notation for reals, 225
- Search menu (VLISP)
  - Bookmarks submenu
    - Clear All Bookmarks command, 41
    - Next Bookmark command, 40
    - Previous Bookmark command, 41
  - Complete Word by Apropos command, 48–49
  - Complete Word by Match command, 47
  - Find command, 37
  - overview, 16
  - Replace command, 39
- search order, for loading program files, 114
- searching
  - Complete Word by Apropos feature, 48–49
  - Complete Word by Match feature, 47–48
  - Console input history, 18, 27
  - Console window, 29
  - for ename associated with handle, 190
  - for file names, 289–290
  - for strings in project source files, 151
  - for symbols using Apropos feature, 43–47
    - Apropos dialog box options, 43–44
    - Copy to Trace/Log feature, 46
    - inserting symbols from results window, 46
    - opening Apropos window, 29, 33, 43
    - overview, 43
    - by prefix, 45
  - searching (*continued*)
    - shortcut menu options, 46–47
    - using search results, 45–47
  - for handle associated with ename, 190
  - for handle associated with object, 191
  - in text editor, 37–39
    - overview, 19
    - repeating earlier searches, 39
    - replacing text, 39
    - shortcut menu command, 33
    - using Find dialog box, 37–39
  - for object associated with handle, 191
- Select All command (Project window shortcut menu), 146
- Select Backward command (VLISP Edit menu), 62
- Select Forward command (VLISP Edit menu), 62
- selected objects in drawings, 107
- selecting
  - copying selected text to new file, 36, 60
  - cursor location and, 34
  - entities, 305, 307–311
  - files to include in projects, 141–144
  - multiple members from Project window, 146–147
  - objects, 305, 307–311
  - running selected lines of code, 21
  - searching highlighted text, 38
  - text before opening files, 34
  - text between cursor and, 62
  - using bookmarks, 41
  - using text editor shortcut keys, 34–35
- selection set data type
  - defined, 226
- selection set handling functions, 296–305, 455
- selection sets
  - adding entities to, 303–304
  - creating, 296–297
  - deleting entities from, 303–304
  - filter lists, 298–304
    - examples, 298–299
    - filtering for extended data, 300
    - logical grouping of filter tests, 302–303
    - overview, 298–300
    - relational tests, 300–302
    - selection set manipulation, 303–304
    - wild-card patterns in, 300
  - finding number of entities, 304
  - passing between AutoLISP and ObjectARX applications, 305
  - returning entity names, 304
  - synopsis of functions for handling, 455
  - testing whether an entity is a member, 304
- semantic auditing of DCL files, 346–347, 387–388
- semicolon (;) VLISP comment styles, 58

- sending commands to AutoCAD prompt, 260–261
- set\_tile function, 372
- setfunhelp function, 290–291
- setq function, 231
- setting
  - breakpoints, 70–71
  - variable values in document namespace, 133
- Setting Case for Symbols options (Format Options dialog box), 57
- setvar function, 263
- sharing data between namespaces, 133–134
- shortcut keys
  - Complete Word by Apropos feature, 48
  - Complete Word by Match feature, 47
  - text editor
    - for code indentation, 36
    - for correcting text, 33
    - for formatting text, 59–60
    - for navigation, 35
    - for selecting text, 34–35
- shortcut menus
  - Apropos results window, 45, 46–47
  - Console window, 29
  - displaying, 29
  - Frame Binding window, 88
  - Inspect window, 102, 104, 105
  - for object line, 105
  - Project window, 146
  - text editor, 32–33
  - Trace Stack window, 87–88
  - Watch window, 84
- single quotation mark ('), forming point lists, 243
- size of dialog boxes, 357
- slashes (//) for DCL comments, 344
- slide\_image function, 383, 384–385
- slider tile, 423
- sliders. *See* tiles (dialog box)
- small\_increment (DCL attribute), 406
- Smart Indent feature, 50, 51
- Snap modes, 274–275
- Sort Block option, 60
- sorting
  - blocks of text, 60
  - variables in Watch window, 84
- Source Position command, Trace Stack window
  - shortcut menu, 88
- spacer tiles (dialog box), 343, 426
- spaces
  - AutoCAD command window versus VLISP
    - Console window, 26
  - in AutoLISP code, 229–230
  - clearing in text editor, 36
  - deleting from cursor to first non-blank character, 60
- spacing in dialog boxes
  - adjusting space at right side or bottom, 350
  - adjusting space between tiles, 349
  - centering buttons, 347–348
  - distributing tiles in a cluster, 348–349
  - fixing spacing around boxed row or column, 350–351
- special characters
  - control characters in DCL strings, 392
  - control characters in quoted strings, 237–238
  - relational operators for selection set filter lists, 300–302
- special forms, 253–254
- special function call frames, 91
- spline-fit polylines, processing, 321
- Split Comments option (Format Options dialog box), 56
- SQL, 472
- .sql files, 140
- ssadd function, 303–304
- ssdel function, 303–305
- ssget function
  - creating selection sets, 296–297
  - restrictions when dialog box is open, 368
  - selection set filter lists, 298–304
- sslength function, 304
- ssmemb function, 304
- ssname function, 304
- st compiler mode parameter, 111
- stack elements
  - calling Inspect feature for, 88
  - defined, 86
  - printing to Console window, 88
  - types of, 86
  - See also* Trace Stack window
- start\_dialog function
  - functions restricted during, 367–368
  - hiding dialog boxes and, 377
  - nested dialog boxes and, 376
  - passing dialog control to AutoCAD, 367
  - Windows and, 367
- start\_image function, 383
- start\_list function, 380–381
- starting
  - Animate mode, 74
  - break loop mode, 75–77
  - debugging session, 76
  - See also* loading: opening; running
  - Trace logging, 82
  - Visual LISP, 12, 22
- status bar (VLISP)
  - asterisk next to file names, 14
  - menu command descriptions on, 14
  - overview, 14
  - toolbar button explanations on, 13

- Step Indicator button, 71
- Step Into command (VLISP Debug menu), 71, 78
- Step Out command (VLISP Debug menu), 78
- Step Over command (VLISP Debug menu), 72, 78
- stepping through programs from breakpoints, 71–72, 78
- Stop Once command (VLISP Debug menu), 75, 76
- storing arbitrary data, 331
- strcase function, 234
- strcat function, 234
- string data type
  - defined, 225
  - overview, 225
- STRING Inspect window, 97, 98
- strings
  - ASCII code conversions, 282–283
  - AutoLISP output functions, 235–239
  - AutoLISP string-handling, 234–235
  - concatenating, 234
  - control characters in quoted strings, 237–238
  - converting angles to, 280
  - converting case, 234
  - converting reals to, 279–280
  - DIESEL string expressions, 266–268
  - displaying messages, 236
  - finding number of characters, 234
  - forcing line breaks in, 237–238
  - returning substrings, 234–235
  - string conversions, 279–281
  - string-handling functions, 444–445
  - stripping file extensions, 234–235
  - wild-card matching, 238–239
- stripping file extensions, 234–235
- strlen function, 234, 235
- stylistic conventions in this manual, 7
- subassemblies. *See* tiles (dialog box)
- SUBR Inspect window, 97, 100
- subst function, 241, 241
- substituting list items, 241, 241
- substr function, 234–235
- Support File Search Path, 111–112
- Symbol command
  - Apropos results window shortcut menu, 46
  - Frame Binding window shortcut menu, 89
  - Watch window shortcut menu, 85
- symbol flags, 94
- SYMBOL Inspect window, 99
- Symbol Service command
  - Console window shortcut menu, 29
  - Inspect window shortcut menu, 102
  - text editor shortcut menu, 33
- Symbol Service dialog box
  - adding symbol to Watch window, 94
  - described, 82
- Symbol Service dialog box (*continued*)
  - displaying function definition, 94
  - help feature, 94
  - opening, 29, 33, 46, 89, 92, 102
  - opening Inspect window, 94
  - overview, 92–93
  - symbol flags, 94
  - toolbar, 94
  - updating symbols, 93
- symbol table
  - names, 240
  - objects, 322–323
  - in the drawing database, 105
- symbolic names for colors, 399–400
- symbols
  - AutoLISP data type, 227–229
  - AutoLISP symbol-handling, 245
  - case setting by VLISP formatter, 57
  - case setting of, 227, 240
  - Debug-on-Entry flag, 75, 94
  - defined, 227
  - finding using Apropos feature, 43–47
    - Apropos dialog box options, 43–44
    - Copy to Trace/Log feature, 46
    - inserting symbols from results window, 46
    - opening Apropos window, 29, 33, 43
    - overview, 43
    - shortcut menu options, 46–47
    - using search results, 45–47
    - with prefix, 45
  - flags, 94
  - inserting from Apropos results window, 46
  - naming restrictions, 227
  - PAUSE symbol, 261–262
  - protected, 228–229
  - restricted characters, 227
  - SYMBOL Inspect window, 97, 99
  - symbol table
    - access functions, 332–333
    - entries that cannot be modified, 322
    - entries that cannot be renamed, 322–323
    - and dictionary handling functions, 455–456
    - objects, 322–323
    - symbol-handling functions, 445
    - using Apropos search results, 45–47
- syntax checking
  - balance of parentheses, 61–63
  - from Project window, 146
  - in text editor, 19
  - using Check command, 64–65
  - using color coding, 63
- Syntax Coloring command (VLISP Tools menu), 42, 43
- :SYNTAX-ERROR trace keyword frame, 91

## system variables

- ERRNO, 486
- specifying values, 263
- TABMODE, 292

## T

T symbol, 232

TAB character in quoted strings, 238

### TAB key

- clearing TAB characters in text editor, 36
- for Console window history, 26–27

tab\_truncate (DCL attribute), 407

tablet function, 291–293

tablets, calibrating, 291–293

TABMODE system variable, 292

tabs (DCL attribute), 406

tblnext function, 332–333

tblsearch function, 332–333

temporary file directory for projects, 154

term\_dialog function, 368, 377, 378

terpri function, 238

text editor, Visual LISP, 31–41

- backing up files automatically, 31–32
- bookmarking text, 40–41
- color coding of files, 19, 42–43, 345
- converting code to comments, 60
- converting comments to active text, 60
- copying text, 36
- described, 3
- editing files, 31
- editing project file source code, 146, 148
- features, 19
- formatting text in active window, 50–51
- indenting or unindenting blocks of text, 59
- loading AutoLISP programs from, 20
- moving text, 36
- need for, 31
- opening
  - existing files, 32
  - LISP programs, 14
  - new files, 31
  - window containing text selected from
    - Inspect window, 103
- overstrike mode, 34
- overview, 14, 19, 31
- replacing text, 39
- restoring from backup files, 32
- running selected lines of code, 21
- saving text to new file, 36, 60
- searching for text, 37–39
- shortcut keys
  - for code indentation, 36
  - for correcting text, 33
  - for formatting text, 59–60
  - for navigation, 35
  - for selecting text, 34–35
- shortcut menu commands, 32–33

text editor, Visual LISP (*continued*)

- sorting text, 60
- starting new lines, 31
- undoing last edit, 31
- window prototyping, 475
- See also* formatting text

text extents, 275–279

text tile, 424

text\_part tile, 424

textbox function, 275–279

textpage function, 267

textscr function, 267

tiles (dialog box)

- adjusting layout, 347–352
  - centering buttons, 347–348
  - customizing exit button text, 351–352
  - distributing tiles in a cluster, 348–349
  - example, 347–348
  - fixing spacing around boxed row or column, 350–351
  - layout guidelines, 357
  - space at right side or bottom, 350
  - space between tiles, 349
- alert boxes, 355, 364
- attributes
  - case-sensitivity, 393
  - data types for values, 392
  - functions handling, 428
  - naming, 393
  - numeric values and, 393
  - overview, 393
  - restricted attributes, 393
  - synopsis of predefined attributes, 394–396
  - syntax, 343–344
  - user-defined attributes, 393–394
- attributes listed by function
  - action expression specification, 397
  - aligning horizontally or vertically in a cluster, 397–426
  - aspect ratio specification, 398
  - background color of image, 399–400
  - bold text specification, 402
  - button is selected with ESC key (or not), 402
  - callback specification, 397
  - character used to mask user input, 406
  - character width units for edit box input, 400
  - default alignment for all tiles in a cluster, 398
  - default button specification, 402–403
  - default height for all tiles in a cluster, 398–399
  - default width for all tiles in a cluster, 399

## tiles (*continued*)

- fixed pitch font for list boxes or pop-up lists, 401
  - initial availability of tile, 403
  - initial choices for list box or pop-up list, 404
  - initial value of tile, 407
  - key of tile to receive initial keyboard focus, 402
  - keyboard mnemonic character for tile, 405
  - maximum characters for edit box input, 400
  - maximum value for slider, 405
  - minimum value for slider, 405
  - multiple selections allowed in list box (or not), 406
  - slider incremental control value specification, 398, 406
  - slider orientation specification, 404
  - specifying if tile is activated, 398
  - tab placement in character width units, 406
  - text larger than tab stop truncated in list box or pop-up list (or not), 407
  - tile height allowed to fill space (or not), 401
  - tile height specification, 401–402
  - tile name specification, 403
  - tile receives focus with TAB key (or not), 403
  - tile text (label) specification, 404
  - tile width allowed to fill space (or not), 401
  - tile width specification, 407
- attributes listed by name, 397–407
- action, 397
  - alignment, 397–426
  - allow\_accept, 398
  - aspect\_ratio, 398
  - big\_increment, 398
  - children\_alignment, 398
  - children\_fixed\_height, 398–399
  - children\_fixed\_width, 399
  - color, 399–400
  - edit\_limit, 400
  - edit\_width, 400
  - fixed\_height, 401
  - fixed\_width, 401
  - fixed\_width\_font, 401
  - height, 401–402
  - initial\_focus, 402
  - is\_bold, 402
  - is\_cancel, 402
  - is\_default, 402–403
  - is\_enabled, 403

## tiles (*continued*)

- is\_tab\_stop, 403
  - key, 403
  - label, 404
  - layout, 404
  - list, 404
  - max\_value, 405
  - min\_value, 405
  - mnemonic, 405
  - multiple\_select, 406
  - password\_char, 406
  - small\_increment, 406
  - tab\_truncate, 407
  - tabs, 406
  - value, 407
  - width, 407
- buttons
- when Cancel and Default button are the same, 352
  - centering, 347–348
  - customizing exit button text, 351–352
  - default\_button definition, 342
  - defining, 342
  - design guidelines, 360–361, 362
  - radio buttons, 360–362
- callback reason codes, 371
- children tiles, 338
- cluster design guidelines, 361
- clusters, handling radio clusters, 374–375
- DCL syntax
- attributes and attribute values, 343–344
  - tile definitions, 342
  - tile references, 343
- defined, 338
- disabling, 358
- edit boxes
- callback reasons, 371
  - design guidelines, 361
  - handling actions and callbacks, 376
- error tiles, 363–364
- functional groupings, 408–410
- decorative and informative tiles, 409
  - dialog box exit buttons and error tile, 410
  - predefined active tiles, 408
  - restricted tiles, 410
  - text clusters, 409
  - tile clusters, 408
- handling, 372–376
- changing modes and values at callback time, 373–374
  - edit boxes, 376
  - initializing modes and values, 372–373
  - mode codes for mode\_tile function, 373

tiles (*continued*)

- radio clusters, 374–375
- sliders, 375–376
- image buttons
  - callback reason code, 371
  - callback reasons, 372
  - color attributes, 384
  - creating images, 383–385
  - design guidelines, 361–362
  - handling, 385
- image tiles
  - color attributes, 384
  - creating images, 383–385
  - design guidelines, 361–362
  - dialog box management, 383–385
  - functions handling, 429
- list boxes
  - appending list items, 381
  - callback reason code, 371
  - changing list items, 381
  - creating lists, 380–381
  - deleting list items, 381
  - design guidelines, 362
  - functions handling, 380, 428
  - handling, 380–382
  - inserting list items, 381
  - list operations, 380–381
  - operation codes for start\_list function, 380
  - processing list elements, 382
- listed by name, 410–426
  - boxed\_column, 410–411
  - boxed\_radio\_column, 411
  - boxed\_radio\_row, 412
  - boxed\_row, 412
  - button, 413
  - column, 413–414
  - concatenation, 414
  - dialog, 414
  - edit\_box, 415
  - errtile, 415
  - image, 416
  - image\_button, 416–417
  - list\_box, 417
  - ok\_cancel, 418
  - ok\_cancel\_help, 418
  - ok\_cancel\_help\_errtile, 418–419
  - ok\_cancel\_help\_info, 419
  - ok\_only, 418
  - paragraph, 419–420
  - popup\_list, 420–421
  - radio\_button, 421
  - radio\_column, 422
  - radio\_row, 422
  - row, 423
  - slider, 423
  - spacer, 425

tiles (*continued*)

- spacer\_0, 426
- spacer\_1, 426
- text, 424
- text\_part, 424
- toggle, 425
- naming, 342, 343
- ok\_cancel tile, 343
- predefined tiles and clusters, 360–364
  - buttons, 360
  - clusters, 361
  - edit boxes, 361
  - image buttons and image tiles, 361–362
  - list boxes, 362
  - radio buttons, radio rows, and radio columns, 360–362
  - sliders, 362
  - text, 363
  - toggles, 363
- prototypes
  - naming, 342
  - overview, 339
  - tile definitions for, 341, 342
- radio buttons, radio rows, and radio columns
  - design guidelines, 360–362
  - handling actions and callbacks, 374–375
- referencing tiles, 343
- sliders
  - callback reason code, 371
  - design guidelines, 362
  - handling actions and callbacks, 375–376
- spacer tiles, 343, 426
- subassemblies
  - children tiles, 338
  - overview, 338
  - tile definitions for, 341
- toggles, design guidelines, 363
- typical tiles, 338
- tiles listed by name
  - spacer\_1, 426
- time
  - changing format for, 60
  - inserting current, 60
- Tmp Directory option (Build Options dialog box), 154
- Toggle Breakpoint command
  - text editor shortcut menu, 33
  - toolbar button, 70
  - VLISP Debug menu, 70, 79
- Toggle Console Log command
  - Console window shortcut menu, 29
  - VLISP File menu, 30
- toggle tile, 425

- Toggle Trace Log command (VLISP File menu), 82
  - toggles (dialog box). *See* tiles (dialog box)
  - toolbars (VLISP)
    - Apropos results window, 46
    - Debug, 70–71
    - Edit, 31
    - Find, 37, 39
    - Project window, 145
    - Run, 20, 21
    - Symbol Service, 94
    - Tools, 50
    - Trace Stack window toolbar, 92
    - Watch, 84
    - See also* specific toolbars
  - Tools menu (AutoCAD), AutoLISP submenu, Visual LISP Editor command, 12
  - Tools menu (VLISP)
    - Check Selection command, 65
    - Check Text in Editor command, 65
    - Environment Options submenu
      - AutoLISP Format Options command, 51, 59
      - General Options command, 32, 75, 104
    - Format Code in Editor command, 50
    - Load Application command, 124
    - Load Text command, 20
    - overview, 17
    - Save Settings command, 58
    - Window Attributes submenu
      - Configure Current command, 42, 79
      - Syntax Coloring command, 42, 43
  - Tools menu (VLISP), Interface Tools submenu, Preview DCL in Editor command, 345
  - Tools toolbar
    - Format Edit window button, 50
    - Format Selection button, 50
    - Help button, 50
  - top forms, 86
  - :TOP-COMMAND trace keyword frame, 89
  - Touch command (Project window shortcut menu), 146
  - Tr symbol flag, 94
  - trace function. *See* AutoLISP Reference
  - Trace log file
    - copying Apropos search results to, 46
    - copying Inspect window contents to, 102
    - copying Watch window contents to, 84
    - turning Trace logging on and off, 82
  - Trace Stack command (VLISP View menu), 87
  - Trace Stack window, 85–92
    - calling Inspect feature for stack elements, 88
    - calling Symbol Service feature for functions, 88
  - Trace Stack window (*continued*)
    - copying stack elements to \*obj\* system variable, 88
    - described, 68, 82
    - displaying Frame Binding window for local variables, 88
    - displaying position of caller expression, 88
    - displaying source text for functions, 88
    - displaying stack element information, 87–88
    - keyword frames, 89–91
    - printing stack elements to Console window, 88
    - refreshing contents, 92
    - shortcut menu commands, 87–88
    - special function call frames, 91
    - stack element lists, 86
    - toolbar, 92
    - trace stack overview, 85–86
    - using the Frame Binding window, 88–92
    - viewing current trace stack, 87
    - viewing error trace stack, 92
    - See also* Frame Binding window
  - Trace symbol flag, 94
  - Trace window
    - copying Apropos search results to, 46
    - copying Inspect window contents to, 102
    - copying Watch window contents to, 84
    - described, 69
    - messages during start-up, 14
    - turning Trace logging on and off, 82
  - Trace window, printing list of arguments to, 206
  - trans function, 286–289
  - transient reactors, 215
  - Transparent option for protected symbols, 228
  - turning on. *See* starting
  - type libraries
    - importing, 193–195
    - using ActiveX without importing, 198–199
  - typographical conventions in this manual, 7
- ## U
- unbalanced parentheses
    - checking in VLISP, 61–63
    - text editor formatter and, 50
  - unbalanced parentheses, checking in AutoLISP, 222–223
  - Uncomment Block option, 60
  - underscore (\_)
    - for foreign-language support, 261
    - for VLISP comments, 58
  - Undo command
    - Console window shortcut menu, 29
    - text editor shortcut menu, 33
    - VLISP Edit menu, 31

- undoing
  - commands issued with command function, 263
  - copy actions, 37
  - in Console window, 29
  - in text editor, 31
  - move actions, 37
- Unindent option, 59
- unit conversions, 283–286
  - inches to meters, 284
  - overview, 283
  - unit definition file, 284–286
- unit definition file, 284–286
  - defining new units, 284
  - derived units, 285–286
  - fundamental units, 284
  - user comments, 286
  - valid name definitions, 284
- unload\_dialog function, 367
- [Un]Select All command (Project window shortcut menu), 146
- Up option (Find dialog box), 37
- Uppcase option, 60
- Update command (Inspect window shortcut menu), 102
- updating
  - ActiveX property with vlax-put-property, 199
  - applications, 125
  - determining if objects available for updating, 183
  - entities, 314–315
  - object properties, 181–183
  - symbols in Symbol Service dialog box, 93
- Use WCMATCH option (Apropos dialog box), 44
- user comments, 286
- User Coordinate System, 286
- User function (Apropos filter value), 44
- user input functions
  - accessing user input from devices, 291
  - allowable input, 268–269
  - arbitrary keyboard input, 273
  - controlling user-input conditions, 271–273
  - getting user input, 268–271
  - getxxx functions, 268–271
  - input options, 271–272
  - keyword options, 272–273
  - pausing for user input, 261–262
  - restrictions when dialog box is open, 367
  - synopsis of, 452
  - validating input, 273
- user-defined DCL attributes, 393–394
- user-defined functions
  - defining with defun function, 245–253
    - adding commands, 248–249
    - c:xxx functions, 247–250

- user-defined functions (*continued*)
  - compatibility with AutoCAD versions, 246–247
  - local variables in functions, 251–252
  - redefining AutoCAD commands, 249–250
  - with arguments, 252–254
  - protected symbols and, 94
  - tracing, 94
- :USER-INPUT trace keyword frame, 89
- USUBR Inspect window, 97, 100

## V

- valid name definitions, 284
- value (DCL attribute), 407
- variables
  - action expression variables, 369–370
  - adding to Watch window, 84
  - assigning values, 227–228, 231
  - AutoLISP, 231–233
  - AutoLISP data type, 227–229
  - blackboard variables, 133–134
  - case sensitivity of, 227
  - clearing from Watch window, 84, 85
  - defined, 227
  - displaying values, 231–232
  - environment variables, inspecting and changing, 264
  - local variables in functions, 251–252
  - localizing, 155, 159, 160
  - monitoring during program execution, 73–74
  - multiple namespaces affect on, 127–128
  - names, case and, 227
  - naming, 228
  - nil variables, 232
  - predefined, 232–233
  - printing values from Watch window, 85
  - referencing in document namespaces, 132–133
  - sorting in Watch window, 84
  - system variables
    - specifying values, 263
    - TABMODE, 292
  - typographical conventions in this manual, 7
  - viewing current value of, 18, 25
- VARIANT Inspect window, 98, 101
- variants
  - changing data type of, 174
  - creating, 174, 174–175
  - creating with an array of four doubles, 179
  - returning data type of, 174
  - returning value of, 174
  - using safearrays with, 178–179
  - working with, 174–175
- vector\_image function, 383



- vertical bar (|) for VLISP comments, 58
- View menu (VLISP)
  - Apropos Window command, 43
  - Browse Drawing Database submenu
    - Browse All Entities command, 104
    - Browse Blocks command, 106
    - Browse Selection command, 107
    - Browse Tables command, 105
    - Inquire Extended Data command, 107
  - overview, 16
  - Trace Stack command, 87
- View Source command (Inspect window shortcut menu), 103
- viewing. *See* displaying; Inspect feature
- Visual LISP
  - closing, 22
  - creating new source files, 229
  - DCL file references, 344
  - exiting, 22
  - extended functions, 22, 457–463
    - collection manipulation functions, 458
    - curve measurement functions, 461–462
    - data conversion functions, 458–459
    - dictionary functions, 462
    - method invocation functions, 459
    - object-handling functions, 463
    - property-handling functions, 460
  - IDE components, 2–3
  - issuing commands from Console window, 14
  - overview, 1–3
  - relationship to AutoCAD, 3
  - relationship to AutoLISP, 1, 3
  - running AutoCAD with, 3, 20
  - starting, 12, 22
  - user interface, 13–15
- Visual LISP Developer's Guide
  - additional publications, 4, 8
  - experience requirements, 4
  - guidelines for using, 4–5
  - organization of, 4–5
  - sections of, 4
  - typographical conventions, 7
- Visual LISP Editor command, AutoCAD Tools menu, 12
- Visual LISP extended functions, 22, 457–463
  - collection manipulation functions, 458
  - curve measurement functions, 461–462
  - data conversion functions, 458–459
  - defined, 22
  - dictionary functions, 462
  - loading extended functions, 22
  - method invocation functions, 459
  - object-handling functions, 463
  - property-handling functions, 460
- Visual LISP extended functions (*continued*)
  - running, 22
  - vl-load-com and, 22
  - VLX namespace functions, 467
- Visual LISP Tutorial, 4
- vla- functions, 170
- vla-get functions, 170, 180
- vla-get-center function, 180, 181
- VLA-objects
  - ActiveX and, 165–169
  - converting between enames and VLA-objects, 190
  - data type, 226
  - defined, 226
- vla-put functions, 170
- vlax- functions, 170
- vlax-3d-point function, 178–179
- vlax-create-object function, 195
- vlax-dump-object function, 185–186
- vlax-ename->vla-object function, 190
- vlax-erased-p function, 183
- vlax-for function, 187–188
- vlax-get-acad-object function, 165
- vlax-get-object function, 195
- vlax-get-property function, 199
- vlax-import-type-library function, 194–195
- vlax-invoke-method function, 198–199
- vlax-make-safearray function, 175–177
- vlax-make-variant function, 174, 174–175
- vlax-map-collection function, 187
- vlax-method-applicable-p function, 186
- vlax-object-released-p function, 189
- vlax-property-available-p function, 186–187
- vlax-put-property function, 199
- vlax-read-enabled-p function, 183
- vlax-release-object function, 189
- vlax-safearray-fill function, 175, 177
- vlax-safearray-put-element function, 177–178
- vlax-variant-change-type function, 174
- vlax-variant-type function, 174
- vlax-variant-value function, 174
- vlax-write-enabled-p function, 183
- vl-bb-ref function, 133
- vl-bb-set function, 133–134
- vl-catch-all-apply, 257
- vl-doc-export function, 130–131, 132, 135
- vl-doc-import function, 132
- vl-doc-ref function, 132, 134
- vl-doc-set function, 132–134
- vl-exit-with-error function, 135
- vl-exit-with-value function, 135
- vlide command, 12
- vlisp command, 12
- VLISP text editor. *See* text editor
- VLISP. *See* Visual LISP
- vlisp-compile function, 110–113, 116
  - compiler modes, 111

- vlisp-compile function (*continued*)
  - example using, 113–114
  - identifying the input file, 112
  - naming an output file, 113
  - overview, 110
  - syntax, 111
- vl-list-exported-function, 131
- vl-list-loaded-vlx function, 131
- vl-load-all function, 129
- vl-load-com function, 165
- vl-load-reactors function, 202, 205
- vl-propagate function, 133
- vlr-add function, 215
- vlr-added-p function, 215
- vlr-beep-reaction function, 206
- vlr-current-reaction-name function, 213
- vlr-data function, 213
- vlr-data-set function, 213
- vlr-owner-add function, 214
- vlr-owner-remove function, 214
- vlr-owners function, 213
- vlr-pers function, 215
- vlr-pers-p function, 215
- vlr-pers-release function, 215
- vlr-reaction-names function, 204–205
- vlr-reactions function, 213
- vlr-reaction-set function, 213
- vlr-reactors function, 211–212
- vlr-remove function, 214
- vlr-remove-all function, 215
- vlr-trace-reaction function, 206
- vlr-type function, 213
- vlr-types function, 204
- vl-unload-vlx function, 129, 131
- .vlx files, 114–117, 118, 123, 125, 139
  - See also* building applications
- VLX namespace functions, 467

## W

- warning messages for safe optimization, 158
- Watch window, 83–85
  - Add Watch button, 73
  - adding symbol from Apropos results window, 47
  - adding symbol from Frame Binding window, 89
  - adding symbol from Symbol Service dialog box, 94
  - adding variables, 84
  - calling Apropos feature, 85
  - calling Symbol Service dialog box, 85
  - clearing all variables, 84
  - copying contents to Trace window, 84

- Watch window (*continued*)
  - copying variable value to \*obj\* system variable, 84
  - described, 3, 68, 82
  - invoking Inspect feature, 84
  - monitoring evaluation results of an expression, 73–74
  - monitoring variables during program execution, 73–74
  - opening, 29, 33
  - overview, 83–84
  - printing variable values, 85
  - removing variables, 85
  - shortcut menu commands, 84
  - sorting variables, 84
  - toolbar, 84

- :WATCH-EVAL trace keyword frame, 89

- wcmatch function, 238–239

- Web site for Autodesk, 8

- Wide style, 53

- width (DCL attribute), 407

- wild-card characters

- in filter lists for selection sets, 300

- in Apropos searches, 44

- matching in strings, 238–239

- Window menu (VLISP), 17

- windows

- Console, 14, 18–19

- controlling AutoCAD graphics and text windows, 267

- copying text from output windows, 15

- for VLISP function output, 15

- menu content variations and, 15–16

- navigating in text editor, 35

- Trace, 14

- Windows Registry functions, 469

- Wizard Mode dialog box, 117

- word completion

- Complete Word by Apropos feature, 48–49

- Complete Word by Match feature, 47–48

- working with existing projects, 149–152

- finding strings in project source files, 151

- including projects in VLISP applications, 151

- opening projects, 149

- World Coordinate System, 286

- World Wide Web site for Autodesk, 8

## X

- X axis of circle, changing, 181–183

- xdata. *See* extended data

- xdroom function, 329

- xdsize function, 329

- xrecord objects, 331