Autodesk.

AutoCAD® 2000

OBJECTARX™ DEVELOPER'S GUIDE

00120-010000-5060 January 19, 1999

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, Drafix, 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.

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.

1 2 3 4 5 6 7 8 9 10

Contents

About ObjectARX Documentation 1

The ObjectARX Documentation Set 2
- Printed Guides 2
- Online Documentation 2
- ObjectARX Logo Program 2
- Where to Start 3

Using This Guide 3
- Organization 3

Part I Using ObjectARX 5

Chapter 1 Overview 7

The ObjectARX Programming Environment 8
- Accessing the AutoCAD Database 8
- Interacting with the AutoCAD Editor 8
- Creating User Interfaces with MFC 9
- Supporting MDI 9
- Creating Custom Classes 9
- Building Complex Applications 9
- Interacting with Other Environments 9

ObjectARX Class Libraries 10
- AcRx Library 10
- AcEd Library 12
- AcDb Library 12
- AcGi Library 13
- AcGe Library 14

Getting Started 16

System Requirements 16
Installing ObjectARX. 16

Chapter 2 Database Primer 19

AutoCAD Database Overview. 20
Multiple Databases 21
Obtaining Object IDs 21
Essential Database Objects 22
Creating Objects in AutoCAD 22
Creating Objects in ObjectARX 25
Creating Entities 25
Creating a New Layer 26
Opening and Closing ObjectARX Objects 27
Adding a Group to the Group Dictionary. 28

Chapter 3 ObjectARX Application Basics. 29

Creating an ObjectARX Application 30
Creating Custom Classes. 31
Responding to AutoCAD Messages 31
Implementing an Entry Point for AutoCAD. 36
Initializing an ObjectARX Application 37
Preparing for Unloading 38
Example Application 39
Registering New Commands 40
Command Stack 40
Lookup Order 42
Global versus Local Command Names 42
Transparent versus Modal Commands 42
Loading an ObjectARX Application 43
The Library Search Path 43
Listing Loaded ObjectARX Applications. 43
Unloading an ObjectARX Application 44
Unlocking Applications. 44
Demand Loading 45
AutoCAD, the Windows System Registry, and ObjectARX Applications 46
Modification of the Registry at ObjectARX Application Installation 47
The DEMANDLOAD System Variable 49
Demand Loading on Detection of Custom Objects 50
Demand Loading on Command 51
Demand Loading on AutoCAD Startup 52
Managing Applications with the System Registry 52
ARX Command. 53

?—List Applications 53
Load 53
Unload 53
Commands 53
Options 53
Running ObjectARX Applications from AutoLISP 55
Error Handling 55

Chapter 4 Database Operations 59

Initial Database 60
Creating and Populating a Database 60
Saving a Database 61
Setting the Default File Format 61
Global Save Functions 63
The wblock Operation 63
Creating a New Database from an Existing Database 63
Creating a New Database with Entities 64
Inserting a Database 65
Setting Current Database Values 66
Database Color Value 66
Database Linetype Value 66
Database Linetype Scale Value 66
Database Layer Value 67
Example of Database Operations 67
Long Transactions 69
Class and Function Overview 69
Long Transaction Example 71
External References 74
External Reference Pre- and Post-Processing 75
File Locking and Consistency Checks 76
Indexes and Filters 77
Drawing Summary Information 79
Last Saved by Autodesk Software 80

Chapter 5 Database Objects 81

Opening and Closing Database Objects 82
Deleting Objects 85
Database Ownership of Objects 85
Adding Object-Specific Data 86
Extended Data 86
Extension Dictionary 89
Erasing Objects 94
Object Filing 95

Chapter 6 Entities . . . 97

Entities Defined . . . 98
Entity Ownership . . . 99
AutoCAD Release 12 Entities . . . 100
Common Entity Properties . . . 101
- Entity Color . . . 101
- Entity Linetype . . . 102
- Entity Linetype Scale . . . 103
- Entity Visibility . . . 104
- Entity Layer . . . 104

Common Entity Functions . . . 105
- Object Snap Points . . . 106
- Transform Functions . . . 107
- Intersecting for Points . . . 107
- GS Markers and Subentities . . . 109
- Exploding Entities . . . 123

Creating Instances of AutoCAD Entities . . . 125
- Creating a Simple Entity . . . 125
- Creating a Simple Block Table Record . . . 126
- Creating a Block Table Record with Attribute Definitions . . . 126
- Creating a Block Reference with Attributes . . . 129
- Iterating through a Block Table Record . . . 133

Complex Entities . . . 134
- Creating a Complex Entity . . . 134
- Iterating through Vertices in a Polyline . . . 135

Coordinate System Access . . . 136
- Entity Coordinate System . . . 136
- AcDb2dPolylineVertex . . . 137

Curve Functions . . . 137
Associating Hyperlinks with Entities . . . 139
- AcDbHyperlink Class . . . 139
- AcDbHyperlinkCollection Class . . . 139
- AcDbEntityHyperlinkPE Class . . . 139
- Hyperlink Example . . . 140

Chapter 7 Container Objects . . . 143

Comparison of Symbol Tables and Dictionaries . . . 144
Symbol Tables . . . 147
- Block Table . . . 149
- Layer Table . . . 149
- Iterators . . . 152

Dictionaries . . . 153
- Groups and the Group Dictionary . . . 153
- MLINE Style Dictionary . . . 156

Layout Dictionary 156
Creating a Dictionary. 157
Iterating over Dictionary Entries 158
Layouts 159
ObjectARX Layout Classes 159
Xrecords. 161
DXF Group Codes for Xrecords 162
Examples 163

Part II User Interfaces 165

Chapter 8 MFC Topics 167

Introduction 168
Using MFC with ObjectARX Applications 168
ObjectARX Applications with Dynamically Linked MFC. 169
Visual C++ Project Settings for Dynamically Linked MFC 169
Debugging ObjectARX Applications with Dynamic MFC. 169
Resource Management. 170
Built-In MFC User Interface Support. 172
Class Hierarchy 173
AdUi Messaging 174
AdUi Tip Windows. 174
AdUi Dialog Classes 174
AcUi Dialog Classes 175
AdUi Classes Supporting Tab Extensibility 176
AdUi and AcUi Control Bar Classes. 176
AdUi and AcUi Edit Controls. 177
AdUi and AcUi Combo Box Controls 178
AcUi MRU Combo Boxes. 179
AdUi Button Classes. 181
AcUi Button Classes 182
Dialog Data Persistency 182
Using and Extending the AdUi Tab Dialog System. 183
Constructing a Custom Tab Dialog That Is Extensible 183
Extending the AutoCAD Built-In Tab Dialogs. 184
Using AdUi and AcUi with VC++ AppWizard. 186
Create the ObjectARX MFC Application Skeleton 186
Create the MFC Dialog Using App Studio 188
Create the Classes and Controls 189
Create the Handlers for the Dialog 190
Add Code to the Handlers 191

Chapter 9 Selection Set, Entity, and Symbol Table Functions 199

Selection Set and Entity Names 200
Handling Selection Sets 200
Selection Set Filter Lists 203
Selection Set Manipulation 209
Transformation of Selection Sets. 211
Entity Name and Data Functions 214
Entity Name Functions 214
Entity Data Functions 223
Entity Data Functions and Graphics Screen 233
Notes on Extended Data 235
Xrecord Objects 241
Symbol Table Access 242

Chapter 10 Global Functions for Interacting with AutoCAD 245

AutoCAD Queries and Commands 246
General Access 246
Getting User Input 258
User-Input Functions. 258
Control of User-Input Function Conditions. 260
Graphically Dragging Selection Sets 263
User Breaks 264
Returning Values to AutoLISP Functions 265
Conversions 266
String Conversions 266
Real-World Units 269
Character Type Handling 270
Coordinate System Transformations 271
Display Control 273
Interactive Output 273
Control of Graphics and Text Screens 275
Control of Low-Level Graphics and User Input 275
Tablet Calibration 276
Wild-Card Matching 278

Part III Defining New Classes 281

Chapter 11 Deriving a Custom ObjectARX Class. 283

Custom Class Derivation 284
Runtime Class Identification 285
Class Declaration Macro 286
Class Implementation Macros 287

Class Initialization Function 289

Chapter 12 Deriving from AcDbObject 291

Overriding AcDbObject Virtual Functions 292
AcDbObject: Essential Functions to Override 292
AcDbObject: Functions Often Overridden 292
AcDbObject: Functions Sometimes Overridden 293
AcDbObject: Functions Rarely Overridden 293
AcRxObject: Functions Rarely Overridden 294
AcDbEntity: Functions to Override 294
AcDbCurve: Functions to Override 295
Implementing Member Functions 297
Filing Objects to DWG and DXF Files. 298
dwgOut() Function 299
dwgIn() Function 299
dxfOut() Function 299
dxfIn() Function 299
Error Checking 300
Implementing the DWG Filing Functions. 300
Implementing the DXF Filing Functions. 302
Object References 310
Ownership References 311
Uses of Ownership 312
Types of Ownership 312
Building an Ownership Hierarchy 312
Pointer References 320
Hard Pointers 320
Soft Pointers 321
Long Transaction Issues for Custom Objects 321
Purge 324
Undo and Redo 324
Automatic Undo. 325
Partial Undo 325
Redo 327
subErase, subOpen, subClose, and subCancel 328
Example of a Custom Object Class 338
Header File 338
Source File 338
Object Version Support 343
Class Versioning. 343
Class Renaming 346
Class Data or Xdata Version Numbers. 347

Chapter 13 Deriving from AcDbEntity 349

Deriving Custom Entities 350
AcDbEntity Functions to Override 350
AcDbEntity Functions Usually Overridden. 351
AcDbEntity Functions Rarely Overridden. 352
Overriding Common Entity Functions 353
Overriding worldDraw() and viewportDraw() 353
Overriding saveAs() 355
Implementing the Object Snap Point Function 357
Implementing the Grip Point Functions 359
Implementing the Stretch Point Functions 361
Transformation Functions. 363
Intersecting with Other Entities 364
Intersecting a Custom Entity with Another Entity. 369
Exploding an Entity 370
Extending Entity Functionality 370
Using AcEdJig 371
Deriving a New Class from AcEdJig 371
General Steps for Using AcEdJig 371
Setting Up Parameters for the Drag Sequence 372
Drag Loop 372
Implementing the sampler(), update(), and entity() Functions . . 375
Adding the Entity to the Database 378
Sample Code 378

Part IV Specialized Topics 385

Chapter 14 Proxy Objects 387

Proxy Objects Defined 388
Proxy Object Life Cycle 388
User Encounters with Proxy Objects 389
Displaying Proxy Entities 390
Editing Proxy Entities 390
Unloading an Application 391

Chapter 15 Notification 393

Notification Overview 394
Reactor Classes. 394
Types of Object Reactors 395
Using Reactors 396
AcDbObject and Database Notification Events 398
Custom Notifications 398

Using an Editor Reactor 398
Using a Database Reactor 399
Using an Object Reactor 402
Notification Use Guidelines 412

Chapter 16 The Multiple Document Interface 415

Overview 416
Document Execution Contexts 416
Data Instances 416
Document Locking 417
Document Management Classes 417
Terminology 418
Active Document 418
Application 418
Application Context 418
Command 418
Command, Multi-Document 419
Command, Nonreentrant 419
Command Processor 419
Current Document 420
Database 420
Document 420
Drawing 420
Edit Session 420
Execution Context, Application 421
MDI-Aware 421
Per-Application 421
Per-Context 421
Per-Document 421
Quiescent 422
Session 422
Undo Stack 422
SDI System Variable 422
Levels of Compatibility 423
SDI-Only Level 423
MDI-Aware Level 424
MDI-Capable Level 427
MDI-Enhanced Level 427
Interacting with Multiple Documents 428
Accessing the Current Document and Its Related Objects 428
Accessing Databases Associated with Noncurrent Documents 429
Setting the Current Document without Activating It 430
Document Event Notification 430
Application-Specific Document Objects 431

Nonreentrant Commands 431
Making a Command Nonreentrant 432
Nonreentrant AutoCAD Commands 432
Multi-Document Commands 432
Disabling Document Switching 435
Application Execution Context 436
Code Invoked under the Application Execution Context 436
Code Differences under the Application Execution Context 436
Other Application Execution Context Considerations. 437
Database Undo and Transaction Management Facilities. 438
Document-Independent Databases 439
An MDI-Aware Example Application 440

Chapter 17 Transaction Management. 449

Overview of Transaction Management 450
Transaction Manager 451
Nesting Transactions 451
Transaction Boundaries 452
Obtaining Pointers to Objects in a Transaction. 453
Newly Created Objects and Transactions. 454
Commit-Time Guidelines. 454
Undo and Transactions 455
Mixing the Transaction Model with the Open and Close Mechanism . . 455
Transactions and Graphics Generation 455
Transaction Reactors. 456
Example of Nested Transactions 457

Chapter 18 Deep Cloning 467

Deep Clone Basics. 468
Using clone() versus deepClone() 468
Key Concepts of Cloning 469
Typical Deep Clone Operation 470
Cloning Objects from Different Owners. 472
Implementing deepClone() for Custom Classes 476
AutoCAD Commands That Use Deep Clone and Wblock Clone 476
Cloning Phase 477
Translation Phase 477
Named Object Dictionary 480
Overriding the deepClone() Function 484
Overriding the wblockClone() Function 488
Using appendAcDbEntity() During Cloning. 498
Handling Hard References to AcDbEntities During wblockClone() 501

Insert 504
Editor Reactor Notification Functions 504

Chapter 19 Protocol Extension 511

Protocol Extension Defined 512
Implementing Protocol Extension 512
Declaring and Defining Protocol Extension Classes 512
Registering Protocol Extension Classes 513
Default Class for Protocol Extension 515
Unloading the Application 515
Using Protocol Extension Functionality in an Application 515
Protocol Extension for the MATCH Command 516
Protocol Extension Example 516

Chapter 20 ObjectARX Global Utility Functions 521

Common Characteristics of ObjectARX Library Functions 522
ObjectARX Global Function Calls Compared to AutoLISP Calls . . 522
Function Return Values versus Function Results. 523
External Functions 524
Error Handling 527
Communication between Applications. 528
Handling External Applications. 532
Variables, Types, and Values Defined in ObjectARX 533
General Types and Definitions 533
Useful Values 539
Result Buffers and Type Codes. 540
ObjectARX Function Result Type Codes 544
User-Input Control Bit Codes 545
Lists and Other Dynamically Allocated Data 546
Result-Buffer Memory Management 548
Extended Data Exclusive Data Types 554
Text String Globalization Issues 555

Chapter 21 Input Point Processing 557

Custom Object Snap Modes 558
Creating and Registering a Custom Object Snap Mode. 558
Creating Protocol Extension Classes 559
Creating a Custom Glyph 561
Custom Object Snap Mode Example 561
Input Point Management 567
Input Point Manager 567

Input Context Events 569
Input Point Filters and Monitors 575

Chapter 22 Application Configuration 585

Profile Manager. 586
AcApProfileManager Class. 586
AcApProfileManagerReactor Class 587
Profile Manager Sample. 588

Part V Interacting with Other Environments 591

Chapter 23 COM, ActiveX Automation, and the Object Property Manager 593

Overview. 594
Using AutoCAD COM Objects from ObjectARX and Other Environments 594
Accessing COM Interfaces from ObjectARX 595
AutoCAD ActiveX Automation Implementation. 605
The Relationship between AcDbObjects and Automation Objects 605
Creating the COM Object 608
Interacting with AutoCAD 611
Document Locking 612
Creating a Registry File. 613
Exposing Automation Functionality 615
Setting Up an ATL Project File 615
Writing a COM Wrapper. 616
Building and Registering a COM DLL. 621
Object Property Manager API 622
AutoCAD COM Implementation 623
Static OPM COM Interfaces 624
ICategorizeProperties Interface 624
IPerPropertyBrowsing Interface. 624
IOPMPropertyExtension Interface 625
IOPMPropertyExpander Interface 625
Implementing Static OPM Interfaces 625
Dynamic Properties and OPM 630
IDynamicProperty 631

Chapter 24 AutoCAD DesignCenter COM API 633

AutoCAD DesignCenter API. 634
IAcDcContentBrowser Interface 634
IAcDcContentView Interface 634

IAcDcContentFinderSite Interface 634
IAcDcContentFinder Interface 635
IAcPostDrop Interface 635
Registry Requirements for an AutoCAD DesignCenter Component 635
Applications Key 635
Extensions Key 636
CLASSID Registration 637
Implementing the Interfaces for AutoCAD DesignCenter 638
Customizing AutoCAD DesignCenter 640
Create an ActiveX Template Library Project 641
Add Registry Support and a New ATL COM Object 641
Add Code to Support the New ATL COM Object 644

Part VI ObjectARX Libraries 653

Chapter 25 The ObjectDBX Libraries 655

Introduction 656
Overview 656
Host Applications 656
ObjectDBX Libraries 656
User Interface and Database Access 657
Using ObjectDBX 657
Getting Started with ObjectDBX 657
ObjectDBX Library Changes 658
The Application Services Class 658
Differences between ObjectDBX and ObjectARX 659
AcEditorReactor Class 660
AcGi API 660
Localization and XMX Files 661
Transaction Management 663
AcTransaction and AcTransactionReactor Classes 663
AcTransactionManager and AcDbTransactionManager Classes . . . 663
Creating a Viewer 663
Viewer Components 664
AcGi 664
AcGix 665
AcGix Differences from AutoCAD Viewing 666
SimpleView 667
WhipView 668
ViewAcDb 669
Basic Viewer Operation 670
Configuration Suggestions 671
Demand Loading 671

Installing the ObjectDBX Libraries 672
Use COMMONFILES 672
Install by Version and as SHAREDFILE 673
Ensure the Files Are on the Path 673
Ensure Smart Pathing Updates 674
Tips and Techniques 676
ACAD_OBJID_INLINE_INTERNAL 676
AcDbDatabase Notes 676
AcDbDatabase::insert() 678
Finding the Active Viewports in Model Space 678
Details About Viewports 679
Always Test Your Drawings in AutoCAD 2000 680
Using DWG Files from Earlier Releases 680
Extended Entity Data 681
Raster Images 682
Known Limitations 682

Chapter 26 The Graphics Interface Library 683

AcGi Overview 684
The setAttributes Function 686
The worldDraw() Function 687
The viewportDraw() Function 688
Viewport Regeneration Type 689
Setting Entity Traits 690
Subentity Traits 691
Useful AcGi Constants 692
Example of Using AcGi 693
Primitives 696
Mesh 696
Shell 700
Arc 703
Polyline 704
Text 704
Using Drawables in Your Object 708
Tessellation 709
Isolines 710
Transformations 710
Model Coordinate System 711
World Coordinate System 712
Eye Coordinate System 712
Display Coordinate System 712
Transformation Examples 712
Using Clip Boundaries in AcGi 723
Background 723

Clip Boundary Example 724

Chapter 27 Using the Geometry Library 725

Overview of the AcGe Library 726
Global Data and Functions 728
Tolerances 729
Using Basic Geometry Types 730
Using the Line and Plane Classes 732
Parametric Geometry 733
Curves 733
Surfaces 736
Special Evaluation Classes 738
Tips for Efficient Use of Curve and Surface Evaluators 744
Persistent AcGe Entities 746
AcGe Persistency Examples 746

Chapter 28 Using the Boundary Representation Library 751

Overview 752
Domain 753
Limitations 754
Class Hierarchy 755
Topological Objects 756
Using Topological Objects in Your Program 757
Using Topological Traversers in Your Program 758
From Topological Traversers to Objects 759
From Mesh Traversers to Mesh Objects 760
AcBr Class Descriptions 761
Entity Classes 761
Containment Classes 762
Mesh Classes 762
Traverser Classes 762
Enumerated Types 764
Error Return Codes 764
Validation Level 764
ShellType 765
LoopType 765
Mesh Element Shape Control 765
Building an Application 765
Sample Application Using the AcBr Library 766

Part VII Appendixes 767

Appendix A Migrating ADS Programs to ObjectARX 769

Migrating to ObjectARX 770
The acrxEntryPoint() Function 770
Header Files 771
Loading Applications: ADS versus ObjectARX 772
Building ADS Applications in the ObjectARX Program Environment . . 773
Sample ObjectARX Application 773
ObjectARX-Exclusive Data Type 778

Appendix B Programmable Dialog Boxes 779

Overview 780
Function Sequence Outline 780
Example Dialog Box 781
Callback Functions 785
Default Actions 786
Passing Arguments in Callback Functions 786
Hiding Dialog Boxes 789
Definitions and Declarations 792
Handles for Dialog Boxes and Tiles 792
Callback Function Definitions 793
Status Codes 793
Handling Tiles 795
Initializing Modes and Values 795
Changing Callback Modes and Values 796
Setting Up List Boxes and Pop-Up Lists 797
Handling List Values 799
Handling Radio Clusters 803
Handling Sliders 804
Handling Edit Boxes 806
Application-Specific Data 806

Index 809

About ObjectARX Documentation

In This Chapter

- The ObjectARX Documentation Set
- Using This Guide

ObjectARX™, the AutoCAD® Runtime Extension programming environment, includes C++ libraries that are the building blocks you can use to develop AutoCAD applications, extend AutoCAD classes and protocol, and create new commands that operate in the same manner as built-in AutoCAD commands.

The ObjectARX documentation set contains both printed and online guides. This chapter gives a brief overview of these guides, and discusses the organization and conventions of the *ObjectARX Developer's Guide*.

The ObjectARX Documentation Set

The ObjectARX documentation set includes the following printed guides and online documentation.

Printed Guides

Two printed manuals are provided with ObjectARX:

- *ObjectARX Developer's Guide*. Explains the concepts of developing an ObjectARX application, with example code and step-by-step procedures.
- *Migration Guide for Applications*. Provides an overview of the new features and changes in the latest versions of the AutoCAD programming environments, including ObjectARX, Visual LISP™, and Visual Basic® for Applications.

Online Documentation

You can access the ObjectARX online documentation from the *\objectarx\docs* directory. The following online documents are provided in Windows help file format:

- *ObjectARX Reference*. A programmer's reference that provides detailed information on each class and function in the ObjectARX API.
- *ObjectARX Developer's Guide*. The same content as the printed *ObjectARX Developer's Guide* in online format.
- *Migration Guide for Applications*. The same content as the printed *Migration Guide for Applications* in online format.
- *ObjectARX Readme*. Describes last-minute changes and additions to ObjectARX.

The online documentation may include updates to the printed material.

ObjectARX Logo Program

Autodesk now offers a "Built with ObjectARX" logo program for AutoCAD applications that use ObjectARX. If you are creating AutoCAD products based on ObjectARX technology, you should look into this program. A guideline for making your application logo-compliant is available from VeriTest, the company that performs the certification process. To find out more about this logo program or to get a copy of the guide, go online to *http://www.veritest.com/autodesk/main(f).htm* and follow the process listed there. Developers with products that meet with the "Built with ObjectARX"

test criteria will be eligible to license and use ObjectARX branding on product packaging, collateral items, and Web sites, and to participate with Autodesk in related marketing initiatives.

NOTE The logo program guidelines contain information about how to register your Registered Developer Symbol (RDS) with Autodesk, in addition to the other logo requirements.

Where to Start

New users should start with the *ObjectARX Developer's Guide*. Experienced users and those upgrading from previous versions of ObjectARX should start with the *Migration Guide for Applications,* and then move on to the more detailed material on the new subjects in the *ObjectARX Developer's Guide*.

Using This Guide

To help you use this book more effectively, the following sections explain the organization of the *ObjectARX Developer's Guide* and the conventions it uses.

Organization

The *ObjectARX Developer's Guide* is organized in seven parts:

Part I: Using ObjectARX describes the fundamental concepts of ObjectARX.

Part II: User Interfaces shows how to work with ObjectARX global functions and MFC to create and interact with user interfaces.

Part III: Defining New Classes describes how to create custom classes in ObjectARX.

Part IV: Specialized Topics examines topics of interest to more advanced users, such as proxy objects, notification, and protocol extension.

Part V: Interacting with Other Environments discusses working with external programming environments such as COM and ActiveX® Automation.

Part VI: ObjectARX Libraries describes several of the ObjectARX libraries, including the ObjectDBX™ libraries and the graphics interface library.

Part VII: Appendixes gives detailed information about migrating ADS programs to ObjectARX, and using programmable dialog boxes.

Part I

Using ObjectARX

Overview

In This Chapter

1

- The ObjectARX Programming Environment
- ObjectARX Class Libraries
- Getting Started

An ObjectARX application is a dynamic link library (DLL) that shares the address space of AutoCAD and makes direct function calls to AutoCAD. You can add new classes to the ObjectARX program environment and export them for use by other programs. The ObjectARX entities you create are virtually indistinguishable from built-in AutoCAD entities. You can also extend the ObjectARX protocol by adding functions at runtime to existing AutoCAD classes.

This chapter provides an overview of the class libraries that compose ObjectARX and gives information for getting started with ObjectARX. The *ObjectARX Developer's Guide* assumes that you are familiar with C++, object-oriented programming, and AutoCAD.

The ObjectARX Programming Environment

The ObjectARX programming environment provides an object-oriented C++ application programming interface for developers to use, customize, and extend AutoCAD. The ObjectARX libraries comprise a versatile set of tools for application developers to take advantage of AutoCAD's open architecture, providing direct access to AutoCAD database structures, the graphics system, and native command definition. In addition, these libraries are designed to work in conjunction with Visual LISP and other application programming interfaces so that developers can choose the programming tools best suited to their needs and experience.

As a developer, you can use ObjectARX to accomplish the following tasks:

- Access the AutoCAD database
- Interact with the AutoCAD editor
- Create user interfaces using the Microsoft® Foundation Classes (MFC)
- Support the multiple document interface (MDI)
- Create custom classes
- Build complex applications
- Interact with other programming environments

The next sections take a brief look at these topics. They will be discussed in greater detail throughout the book.

Accessing the AutoCAD Database

An AutoCAD drawing is a collection of objects stored in a database. These objects represent not only graphical entities, but also internal constructs such as symbol tables and dictionaries. ObjectARX provides your application with access to these database structures. In addition, you can create new database objects for your specific application.

Interacting with the AutoCAD Editor

ObjectARX provides classes and member functions to interact with the AutoCAD editor. You can register commands with AutoCAD that will be treated as built-in commands. Your application can receive and respond to notification about a variety of events that occur within AutoCAD.

Creating User Interfaces with MFC

ObjectARX applications can be built with a dynamically linked MFC library that is shared with AutoCAD. You can use this library to create standard Microsoft Windows graphical user interfaces (GUIs).

Supporting MDI

With ObjectARX, you can create applications that will support the AutoCAD multiple document interface, and you can ensure that your applications will interact properly with other applications in the Microsoft Windows environment.

Creating Custom Classes

You can leverage the classes in the ObjectARX hierarchy to create your own custom classes. In addition, you can make use of the extensive graphics libraries of ObjectARX when creating custom classes.

Building Complex Applications

ObjectARX supports the development of complex applications, providing the following features:

- Notification
- Transaction management
- Deep cloning
- Reference editing
- Protocol extension
- Proxy object support

Interacting with Other Environments

ObjectARX applications can communicate with other programming interfaces, such as Visual LISP, ActiveX, and COM. In addition, ObjectARX applications can interact with the Internet, by associated URLs with entities, and by loading and saving drawing files from the World Wide Web (WWW).

ObjectARX Class Libraries

The ObjectARX environment consists of the following groups of classes and functions:

AcRx	Classes for binding an application and for runtime class registration and identification.
AcEd	Classes for registering native AutoCAD commands and for AutoCAD event notification.
AcDb	AutoCAD database classes.
AcGi	Graphics classes for rendering AutoCAD entities.
AcGe	Utility classes for common linear algebra and geometric objects.

The following table lists the libraries required to link ObjectARX applications. All ObjectARX applications must link with *acad.lib* and *rxapi.lib*. Other libraries may also be required, depending on the prefix of the ObjectARX classes and functions that you are using.

Required ObjectARX libraries

Prefix	Required Libraries
AcRx	acad.lib, rxapi.lib, acrx15.lib
AcEd	acad.lib, rxapi.lib, acedapi.lib, acrx15.lib
AcDb	acad.lib, rxapi.lib, acdb15.lib, acrx15.lib
AcGi	acad.lib, rxapi.lib, acgiapi.lib, acrx15.lib
AcGe	acad.lib, rxapi.lib, acge15.lib, acrx15.lib

The following sections take a closer look at each of the ObjectARX libraries. For more information about specific classes and member functions, see the *ObjectARX Reference*.

AcRx Library

The AcRx library provides system-level classes for DLL initialization and linking and for runtime class registration and identification. The base class of this library is `AcRxObject`, which provides the following facilities:

- Object runtime class identification and inheritance analysis
- Runtime addition of new protocol to an existing class (see chapter 19, "Protocol Extension")
- Object equality and comparison testing
- Object copy

The AcRx library also provides a set of C++ macros to help you create new ObjectARX classes derived from `AcRxObject` (see chapter 11, "Deriving a Custom ObjectARX Class").

`AcRxDictionary` is another important class in this library. A dictionary is a mapping from a text string to another object. The AcRx library places its objects, classes, and service dictionaries in a global object dictionary, which is an instance of the `AcRxDictionary` class. Applications can add objects to this dictionary so that they are accessible to other applications.

The class hierarchy for the AcRx library is as follows:

```
AcRxObject
    ├─ AcRxClass
    ├─ AcRxDictionary
    ├─ AcRxDynamicLinker
    ├─ AcRxEvent
    │   └─ AcEditor
    ├─ AcRxService
    │   ├─ AcRxKernal
    │   ├─ AcDbServices
    │   └─ AcEdServices
    └─ AcadAppInfo
```

Runtime Type Identification

Every subclass of `AcRxObject` has an associated class descriptor object (of type `AcRxClass`) that is used for runtime type identification. ObjectARX provides functions for testing whether an object is of a particular class or derived class, functions for determining whether two objects are of the same class, and functions for returning the class descriptor object for a given class.

For more information on using AcRx classes, see chapter 3, "ObjectARX Application Basics," chapter 11, "Deriving a Custom ObjectARX Class," and chapter 19, "Protocol Extension."

AcEd Library

The AcEd library provides classes for defining and registering new AutoCAD commands that operate in the same manner as built-in AutoCAD commands. The new commands you define are referred to as "native" commands because they reside in the same internal structure (the `AcEdCommandStack`) as built-in commands. The AcEd library also provides an editor reactor and a set of global functions for interacting with AutoCAD. An important class in this library is `AcEditorReactor`; it monitors the state of the AutoCAD editor and notifies the application when specified events occur, such as starting, ending, or canceling a command.

The class hierarchy for the AcEd library is as follows:

```
AcRxObject
    ├─ AcEdCommand
    ├─ AcEdCommandStack
    ├─ AcEdUIContext
    ├─ AcEdJig
    ├─ AcEdInputPointFilter
    ├─ AcEdInputPointMonitor
    │   ├─ AcEdInputPointManager
    │   └─ AcEdSolidSubentitySelector
    │
    ├─ AcTransaction
    └─ AcDbTransactionManager
        └─ AcTransactionManager
```

For information on registering new AutoCAD commands using ObjectARX, see chapter 3, "ObjectARX Application Basics." For an example of using an editor reactor, see chapter 15, "Notification."

AcDb Library

The AcDb library provides the classes that compose the AutoCAD database. This database stores all the information for the graphical objects, called *entities*, that compose an AutoCAD drawing, as well as the nongraphical objects (for example, layers, linetypes, and text styles) that are also part of a drawing. You can query and manipulate existing instances of AutoCAD entities and objects with the AcDb library, and you can create new instances of database objects.

The AutoCAD database contains these major elements:

- A set of nine symbol tables that own uniquely named symbol table entry objects. These objects represent various commonly used `AcDbDatabase` objects and data members.

- A named object dictionary (of class `AcDbDictionary`), which provides the "table of contents" for an AutoCAD drawing. Initially, this table of contents contains the IDs of the four other dictionaries used by AutoCAD. Applications you develop, however, are free to add other objects to the dictionary.
- A fixed set of about 200 header variables, whose values are set by AutoCAD.

The class hierarchy for the AcDb library is as follows:

```
AcRxObject
  ├─ AcDbDictionary
  │    └─ AcDbDictionaryWithDefault
  ├─ AcDbFilter
  │    ├─ AcDbLayerFilter
  │    └─ AcDbSpatialFilter
  ├─ AcDbGroup
  ├─ AcDbIDBuffer
  ├─ AcDbIndex
  │    ├─ AcDbLayerIndex
  │    └─ AcDbSpatialIndex
  ├─ AcDbLongTransaction
  ├─ AcDbMlineStyle
  ├─ AcDbPlaceholder
  ├─ AcDbPlotSettings
  │    └─ AcDbLayout
  ├─ AcDbProxyObject
  ├─ AcDbXrecord
  └─ AcDbEntity
       ├─ AcDbSymbolTable
       │    └─ AcDbAbstractViewTable
       │         ├─ AcDbViewportTable
       │         └─ AcDbViewTable
       ├─ AcDbBlockTable
       ├─ AcDbDimStyleTable
       ├─ AcDbFontTable
       ├─ AcDbLayerTable
       ├─ AcDbLinetypeTable
       ├─ AcDbRegAppTable
       ├─ AcDbTextStyleTable
       ├─ AcDbUCSTable
       ├─ AcDbSymbolTableRecord
       │    └─ AcDbAbstractViewTableRecord
       │         ├─ AcDbViewportTableRecord
       │         └─ AcDbViewTableRecord
       ├─ AcDbBlockTableRecord
       ├─ AcDbDimStyleTableRecord
       ├─ AcDbFontTableRecord
       ├─ AcDbLayerTableRecord
       ├─ AcDbLinetypeTableRecord
       ├─ AcDbRegAppTableRecord
       ├─ AcDbTextStyleTableRecord
       ├─ AcDbUCSTableRecord
       ├─ AcDbRasterImageDef
       ├─ AcDbRasterImageDefReactor
       └─ AcDbRasterVariables
```

For more information on the AcDb library, see chapter 2, "Database Primer," chapter 4, "Database Operations," chapter 5, "Database Objects," chapter 6, "Entities," and chapter 7, "Container Objects." For information on deriving new classes from `AcDbObject` and `AcDbEntity`, see chapter 12, "Deriving from AcDbObject" and chapter 13, "Deriving from AcDbEntity."

AcGi Library

The AcGi library provides the graphics interface used for drawing AutoCAD entities. This library is used by the `AcDbEntity` member functions `worldDraw()`, `viewportDraw()`, and `saveAs()`, all part of the standard entity protocol. The `worldDraw()` function must be defined by all custom entity classes. The `AcGiWorldDraw` object provides an API through which `AcDbEntity::worldDraw()` can produce its graphical representation in all viewports simultaneously. Similarly, the `AcGiViewportDraw` object provides an API through which the `AcDbEntity::viewportDraw()` function can produce different graphical representations for each viewport.

The class hierarchy for the AcGi library is as follows:

```
AcRxObject
    ├─ AcGiCommonDraw
    │   ├─ AcGiWorldDraw
    │   └─ AcGiWorldDraw
    ├─ AcGiContext
    ├─ AcGiEdgeData
    ├─ AcGiFaceData
    ├─ AcGiGeometry
    │   ├─ AcGiViewportGeometry
    │   └─ AcGiWorldGeometry
    ├─ AcGiLinetypeEngine
    ├─ AcGiSubEntityTraits
    │   └─ AcGiDrawableTraits
    ├─ AcGiTextStyle
    ├─ AcGiVertexData
    ├─ AcGiViewport
    └─ AcGiDrawable
        └─ AcGiGlyph
```

For more information on using AcGi classes, see chapter 13, "Deriving from AcDbEntity."

AcGe Library

The AcGe library is used by the AcDb library and provides utility classes such as vectors and matrices that are used to perform common 2D and 3D geometric operations. It also provides basic geometric objects such as points, curves, and surfaces.

The AcGe library consists of two major subsets: classes for 2D geometry and classes for 3D geometry. The major abstract base classes are `AcGeEntity2d` and `AcGeEntity3d`. Several basic classes not derived from any other class include `AcGePoint2d`, `AcGeVector2d`, and `AcGeMatrix2d` (shown at the beginning of the class hierarchy). These basic classes can be used to perform many types of common operations, such as adding a vector to a point, computing the dot or cross product of two vectors, and computing the product of two matrices. The higher-level classes of this library are implemented using these basic classes. The class hierarchy for the AcGe library is as follows:

```
AcGeEntity2d
├─ AcGeBoundBlock2d
├─ AcGeClipBoundary2d
├─ AcGeCurve2d
│   ├─ AcGeCircArc2d
│   ├─ AcGeCompositeCurve2d
│   ├─ AcGeEllipArc2d
│   ├─ AcGeExternalCurve2d
│   ├─ AcGeLinearEnt2d
│   │   ├─ AcGeLine2d
│   │   ├─ AcGeLineSeg2d
│   │   └─ AcGeRay2d
│   ├─ AcGeOffsetCurve2d
│   └─ AcGeSplineEnt2d
│       ├─ AcGeCubicSplineCurve2d
│       ├─ AcGeNurbCurve2d
│       └─ AcGePolyline2d
├─ AcGeCurveCurveInt2d
└─ AcGePointEnt2d
    ├─ AcGePointOnCurve2d
    └─ AcGePosition2d

AcGeCurveBoundary
AcGe
AcGeContext
AcGeDwgIO
AcGeDxfIO
AcGeFileIO
AcGeFiler
AcGeInterval
AcGeKnotVector
AcGeLibVersion
AcGeMatrix2d
AcGeMatrix3d
AcGePoint2d
└─ AcAxPoint2d
AcGePoint3d
└─ AcAxPoint3d
AcGeScale2d
AcGeScale3d
AcGeTol
AcGeVector2d
AcGeVector3d
```

```
AcGeEntity3d
├─ AcGeBoundBlock3d
├─ AcGeCurve3d
│   ├─ AcGeCircArc3de
│   ├─ AcGeCompositeCurve3d
│   ├─ AcGeEllipArc3e
│   ├─ AcGeExternalCurve3d
│   ├─ AcGeLinearEnt3d
│   │   ├─ AcGeLine3d
│   │   ├─ AcGeLineSeg3d
│   │   ├─ AcGeRay3d
│   │   └─ AcGeMatrix3d
│   ├─ AcGeOffsetCurve3d
│   └─ AcGeSplineEnt3d
│       ├─ AcGeCubicSplineCurve3d
│       ├─ AcGeNurbCurve3d
│       └─ AcGePolyline3d
│           └─ AcGeAugPolyline3d
├─ AcGeCurveCurveInt3d
├─ AcGeCurveSurfInt
├─ AcGePointEnt3d
│   ├─ AcGePointOnCurve3d
│   ├─ AcGePointOnSurface
│   └─ AcGePosition3d
├─ AcGeSurfSurfInt
└─ AcGeSurface
    ├─ AcGeCone
    ├─ AcGeCylinder
    ├─ AcGeExternalBoundedSurface
    ├─ AcGeExternalSurface
    ├─ AcGeNurbSurface
    ├─ AcGeOffsetSurface
    ├─ AcGePlanarEnt
    │   ├─ AcGeBoundedPlanet
    │   └─ AcGePlane
    ├─ AcGeSphere
    └─ AcGeTorus
```

The AcGe library provides several different coordinate systems. For more information, see chapter 27, “Using the Geometry Library.” The sample programs in this manual illustrate numerous common uses of AcGe classes.

Getting Started

The following sections discuss the system requirements for ObjectARX and provide installation instructions.

System Requirements

Developing applications with ObjectARX requires the following software and hardware:

- Windows NT® 4.0
- Microsoft Visual C++® 32bit Edition Release 6.0
- Pentium® PC running at 90MHz or better, with 32MB RAM or more
- 800 x 600 SVGA display or better

Installing ObjectARX

When you install ObjectARX, a setup program guides you through the process.

To install ObjectARX

1. Insert the CD into the CD-ROM drive.
2. If you are running Windows NT 4.0 with AutoPlay, follow the on-screen instructions.
3. If you have turned off AutoPlay in Windows NT 4.0, from the Start menu on the taskbar, choose Run, designate the CD-ROM drive, enter the path name, and then enter **setup**.

ObjectARX Directory Tree

The default installation for ObjectARX software creates the base directory *c:\objectarx*. The nine main subdirectories under the *objectarx* directory are described here.

arxlabs — The *arxlabs* directory consists of a set of subdirectories, each containing a lab (tutorial) that demonstrates implementation of one aspect of the ObjectARX API. Each subdirectory includes instructions for the lab, a source code file with key pieces missing that are to be filled in by the user according to the instructional comments in the code, and a *solved* subdirectory containing the completed source code file for the lab.

classmap	The *classmap* directory contains an AutoCAD drawing illustrating the ObjectARX class hierarchy.
docs	The *docs* directory contains Windows online help files for ObjectARX developers, including the *ObjectARX Developer's Guide*, the *ObjectARX Reference*, the *Migration Guide for Applications*, and the *ObjectARX Readme* file.
docsamps	The *docsamps* directory contains subdirectories for each of the programs from which examples were extracted for the *ObjectARX Developer's Guide*. Each subdirectory contains the full set of source code for the application and an explanatory *Readme* file.
inc	The *inc* directory contains the ObjectARX header files.
lib	The *lib* directory contains the ObjectARX library files.
redistrib	The *redistrib* directory contains a set of DLLs, some of which may be required for an ObjectARX application to run. Developers should copy the DLLs that they need for application development to a directory in the AutoCAD search path, and package the necessary DLLs with their ObjectARX applications for distribution.
samples	The *samples* directory includes subdirectories containing examples of ObjectARX applications. These subdirectories include source code and *Readme* files. The most significant set of sample ObjectARX applications is in the *polysamp* subdirectory.
utils	The *utils* directory contains subdirectories for applications that are extensions to ObjectARX, including *brep* for boundary representation and *istorage* for compound document storage. Each application directory includes *inc*, *lib*, and *sample* subdirectories.

Database Primer

In This Chapter

2

- AutoCAD Database Overview
- Essential Database Objects
- Creating Objects in AutoCAD
- Creating Objects in ObjectARX

The AutoCAD database stores the objects and entities that make up an AutoCAD drawing. This chapter discusses the key elements of the database: entities, symbol tables, and the named object dictionary.

This chapter also introduces object handles, object IDs, and the protocol for opening and closing database objects. Sample code gives an example of creating entities, layers, and groups, and adding objects to the database.

AutoCAD Database Overview

An AutoCAD drawing is a collection of objects stored in a database. Some of the basic database objects are entities, symbol tables, and dictionaries. Entities are a special kind of database object that have a graphical representation within an AutoCAD drawing. Lines, circles, arcs, text, solids, regions, splines, and ellipses are examples of entities. A user can see an entity on the screen and can manipulate it.

Symbol tables and dictionaries are containers used to store database objects. Both container objects map a symbol name (a text string) to a database object. An AutoCAD database includes a fixed set of symbol tables, each of which contains instances of a particular class of symbol table record. You cannot add a new symbol table to the database. Examples of symbol tables are the layer table (`AcDbLayerTable`), which contains layer table records, and the block table (`AcDbBlockTable`), which contains block table records. All AutoCAD entities are owned by block table records.

Dictionaries provide a more generic container for storing objects than symbol tables. A dictionary can contain any object of the type `AcDbObject` or subclass thereof. The AutoCAD database creates a dictionary called the named object dictionary when it creates a new drawing. The named object dictionary can be viewed as the master "table of contents" for all of the dictionaries associated with the database. You can create new dictionaries within the named object dictionary and add new database objects to them.

The following figure shows the key components of the AutoCAD database.

During AutoCAD edit sessions, you can obtain the database for the current drawing by calling the following global function:

```
acdbHostApplicationServices()->workingDatabase()
```

Multiple Databases

Multiple databases can be loaded in a single AutoCAD session. Each object in the session has a handle and an object ID. A handle uniquely identifies the object within the scope of a particular database, whereas an object ID uniquely identifies the object across all databases loaded at one time. An object ID only persists during an edit session, but a handle gets saved with the drawing. In contrast to the object ID, an object handle is not guaranteed to be unique when multiple databases are loaded in an AutoCAD session.

Obtaining Object IDs

With an object ID, you can obtain a pointer to an actual database object so that you can perform operations on it. For an example, see "Opening and Closing ObjectARX Objects" on page 27.

You can obtain an object ID in a number of ways:

- Create an object and append it to the database. The database then gives the object an ID and returns it to you.
- Use the database protocol for obtaining the object ID of the objects that are created automatically when a database is created (such as the fixed set of symbol tables and the named object dictionary).
- Use class-specific protocol for obtaining object IDs. Certain classes, such as symbol tables and dictionaries, define objects that own other objects. These classes provide protocol for obtaining the object IDs of the owned objects.
- Use an iterator to step through a list or set of objects. The AcDb library provides a number of iterators that can be used to step through various kinds of container objects (`AcDbDictionaryIterator`, `AcDbObjectIterator`).
- Query a selection set. After the user has selected an object, you can ask the selection set for the list of entity names of the selected objects, and from the names convert to the object IDs. For more information on selection sets, see chapter 6, "Entities."

Essential Database Objects

As objects are created in AutoCAD, they are added to their appropriate container object in the database. Entities are added to the records in the block table. Symbol table records are added to the appropriate symbol tables. All other objects are added to the named object dictionary or to objects that are owned by other objects (and, ultimately, by the named object dictionary), or to an extension dictionary. The scenario in the following section, "Creating Objects in AutoCAD," details this process. Extension dictionaries are discussed in the section "Extension Dictionary" on page 89.

To be usable, a database must have at least the following set of objects:

- A set of nine symbol tables that includes the block table, layer table, and linetype table. The block table initially contains three records: a record called *MODEL_SPACE, and two paper space records called *PAPER_SPACE and *PAPER_SPACE0. These block table records represent model space and the two predefined paper space layouts. The layer table initially contains one record, layer 0. The linetype table initially contains the CONTINUOUS linetype.
- A named object dictionary. When a database is created, this dictionary already contains four database dictionaries: the GROUP dictionary, MLINE style dictionary, layout dictionary, and plot style name dictionary. Within the MLINE style dictionary, the STANDARD style is always present.

These objects can be automatically created in a new database by passing `kTrue` in for its constructor's `buildDefaultDrawing` argument. Passing in `kFalse` creates an empty database into which a DWG or DXF™ file can be loaded.

```
AcDbDatabase(Adesk::Boolean buildDefaultDrawing = Adesk::kTrue);
```

Creating Objects in AutoCAD

This section describes creating a line, circle, layer, and group in AutoCAD and shows how AutoCAD adds these objects to the database. First, suppose the user creates a line in model space with the following command:

line 4,2 10,7

In the database, AutoCAD creates an instance of class `AcDbLine` and then stores it in the model space block table record as shown in the following illustration:

When you first invoke AutoCAD and the database is in its default state, entities are added to model space, the main space in AutoCAD, which is used for model geometry and graphics. Paper space is intended to support "documentation" geometry and graphics, such as drafting sheet outlines, title blocks, and annotational text. The entity creation commands in AutoCAD (LINE, in this case) cause the entity to be added to the current database as well as to the model space block. You can ask any entity which database and which block it belongs to.

Next, suppose the user creates a circle with this command:

circle 9,3 2

Again, AutoCAD creates an instance of the appropriate entity—here, `AcDbCircle`—and adds it to the model space block table record.

Next, the user creates a layer:

layer _make mylayer

AutoCAD creates a new layer table record to hold the layer and then adds it to the layer table.

Finally, the user groups all the entities together:

group 3,2 9,3

AutoCAD creates a new group object and adds it to the GROUP dictionary, which is contained in the named object dictionary. The new group contains a list of the object IDs of the objects that compose the group.

Creating Objects in ObjectARX

The sample ObjectARX code in this section creates the same entities as those in the previous section (a line and a circle). Code for creating a new layer, changing the color of the line, and adding a group to the GROUP dictionary are also shown.

Creating Entities

The following ObjectARX code creates the line and adds it to the model space block table record:

```
AcDbObjectId
createLine()
{
    AcGePoint3d startPt(4.0, 2.0, 0.0);
    AcGePoint3d endPt(10.0, 7.0, 0.0);
    AcDbLine *pLine = new AcDbLine(startPt, endPt);

    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);
    pBlockTable->close();

    AcDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);

    pBlockTableRecord->close();
    pLine->close();

    return lineId;
}
```

The `createLine()` routine obtains the block table for the current drawing. Then it opens the model space block table record for writing. After closing the block table, it adds the entity to the block table record and then closes the block table record and the entity.

NOTE When you are done using any ObjectARX objects, you must explicitly close them as soon as possible.

The following `createCircle()` routine creates the circle and adds it to the model space block table record:

```
AcDbObjectId
createCircle()
{
    AcGePoint3d center(9.0, 3.0, 0.0);
    AcGeVector3d normal(0.0, 0.0, 1.0);
    AcDbCircle *pCirc = new AcDbCircle(center, normal, 2.0);

    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);
    pBlockTable->close();

    AcDbObjectId circleId;
    pBlockTableRecord->appendAcDbEntity(circleId, pCirc);

    pBlockTableRecord->close();
    pCirc->close();

    return circleId;
}
```

Creating a New Layer

The following code obtains the layer symbol table from the database, creates a new layer table record, and names it (`ASDK_MYLAYER`). The layer table record is then added to the layer table.

```
void
createNewLayer()
{
    AcDbLayerTable *pLayerTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pLayerTable, AcDb::kForWrite);

    AcDbLayerTableRecord *pLayerTableRecord =
        new AcDbLayerTableRecord;
    pLayerTableRecord->setName("ASDK_MYLAYER");

    // Defaults are used for other properties of
    // the layer if they are not otherwise specified.
    //
    pLayerTable->add(pLayerTableRecord);
    pLayerTable->close();
    pLayerTableRecord->close();
}
```

Opening and Closing ObjectARX Objects

All code examples shown in this chapter illustrate the protocol for opening and closing objects that you'll need to observe whenever you work with database-resident objects. This protocol ensures that objects are physically in memory when they need to be accessed but can be paged out to disk when they're not needed. Before you can modify an object, you need to open it, as shown in the following example:

```
acdbOpenObject(pObject, objId, AcDb::kForWrite);
```

The open functions have a mode parameter that specifies whether you are opening the object for read, write, or notify. While the object is open for write, you can modify it. When you are finished, you must explicitly close the object as shown in the following example, regardless of the mode in which it was opened:

```
pObject->close();
```

The following is sample code for changing the color of an entity:

```
Acad::ErrorStatus
changeColor(AcDbObjectId entId, Adesk::UInt16 newColor)
{
    AcDbEntity *pEntity;
    acdbOpenObject(pEntity, entId,
        AcDb::kForWrite);

    pEntity->setColorIndex(newColor);
    pEntity->close();

    return Acad::eOk;
}
```

New instances of an object are considered to be open for write. Some functions, such as the `AcDbBlockTable::getAt()` function, obtain an object ID and open the object at the same time. An object can't be closed until it has been added to the database. You own the object and can freely delete it at any time before the object is added to the database.

However, once the object has been added to the database, you cannot delete it directly. You can call the `AcDbObject::erase()` function, which marks the object as erased. Erased objects remain in the database until the database is destroyed, but do not get saved when the drawing is saved.

WARNING! Directly deleting an object that has been added to the database will cause AutoCAD to terminate.

Adding a Group to the Group Dictionary

The following code creates a group (`pGroup`) out of the line and circle created in the `createLine()` and `createCircle()` functions and puts the group into the GROUP dictionary. The object IDs of the line and circle are passed into the function. Notice how the GROUP dictionary is opened for writing, modified, and then explicitly closed.

```
void
createGroup(AcDbObjectIdArray& objIds, char* pGroupName)
{
    AcDbGroup *pGroup = new AcDbGroup(pGroupName);
    for (int i = 0; i < objIds.length(); i++) {
        pGroup->append(objIds[i]);
    }

    // Put the group in the group dictionary that resides
    // in the named object dictionary.
    //
    AcDbDictionary *pGroupDict;
    acdbHostApplicationServices()->workingDatabase()
        ->getGroupDictionary(pGroupDict, AcDb::kForWrite);

    AcDbObjectId pGroupId;
    pGroupDict->setAt(pGroupName, pGroup, pGroupId);
    pGroupDict->close();
    pGroup->close();
}
```

ObjectARX Application Basics

In This Chapter 3

- Creating an ObjectARX Application
- Example Application
- Registering New Commands
- Loading an ObjectARX Application
- Unloading an ObjectARX Application
- Demand Loading
- ARX Command
- Running ObjectARX Applications from AutoLISP
- Error Handling

This chapter describes how to write and run an ObjectARX application. It lists the messages passed by AutoCAD to the ObjectARX application and shows how the application typically responds to those messages. This chapter also discusses the registration of new commands, how to load and unload an application, AutoCAD's demand loading feature, and error handling.

Creating an ObjectARX Application

An ObjectARX application is a DLL that shares AutoCAD's address space and makes direct function calls to AutoCAD. ObjectARX applications typically implement commands that can be accessed from within AutoCAD. These commands are often implemented using custom classes. Creating an ObjectARX application involves the following general steps.

To create an ObjectARX application

1. Create custom classes to implement new commands.

   You can derive custom classes from most of the ObjectARX hierarchy and symbol table classes.

2. Determine which AutoCAD messages your ObjectARX application will handle.

   AutoCAD sends a variety of messages to ObjectARX applications, indicating that particular events have occurred within AutoCAD. You decide which messages your application will respond to, and which actions will be triggered.

3. Implement an entry point for AutoCAD.

   AutoCAD calls into an ObjectARX application through the `acrxEntryPoint()` function, which replaces the `main()` function of a C++ program. You are responsible for implementing the `acrxEntryPoint()` function in your application. The `acrxEntryPoint()` function calls the functions that you've associated with specific AutoCAD messages.

4. Implement initialization.

   Within your ObjectARX application, you will need to initialize any custom classes that you have created, and rebuild the ObjectARX runtime class tree. Additionally, if you are adding commands, you must register them with AutoCAD.

5. Prepare for unloading.

   To create a well-behaved ObjectARX application, you must remove any custom classes and commands when your application is unloaded.

   The following sections discuss the general steps of developing an ObjectARX application in more detail.

NOTE An ObjectARX Wizard is available for creating ObjectARX projects. See the *objectarx\utils* directory in the ObjectARX SDK.

Creating Custom Classes

You can derive custom classes from most of the ObjectARX class hierarchy. This allows you to leverage the functionality of the ObjectARX classes when creating your own objects. Defining custom classes is discussed in detail in chapter 11, "Deriving a Custom ObjectARX Class."

Responding to AutoCAD Messages

There are four categories of messages that AutoCAD sends to ObjectARX applications:

- Messages that are sent to all applications
- Messages that are sent only if the application has registered an AutoLISP® function with `acedDefun()`
- Messages that are sent to applications that have registered a service with ObjectARX
- Messages only responded to by applications that use ActiveX Automation

The following five tables describe the messages that AutoCAD sends to ObjectARX applications. The first table lists messages sent to all applications.

Messages sent to all applications

Message	Description
kInitAppMsg	Sent when the ObjectARX application is loaded to open communications between AutoCAD and the application.
kUnloadAppMsg	Sent when the ObjectARX application is unloaded (either when the user unloads the application or when AutoCAD itself is terminated). Closes files and performs cleanup operations.
kLoadDwgMsg	Sent once when the drawing is opened. Then, if the application registers any functions with AutoLISP, AutoCAD sends this message once for each drawing loaded into the editor. The AutoCAD editor is fully initialized at this point, and all global functions are available. However, you cannot use an acedCommand() function from a kLoadDwgMsg.
kPreQuitMsg	Sent when AutoCAD quits, but before it begins to unload all ObjectARX applications.

The next table lists messages that AutoCAD sends to applications that have registered an AutoLISP function with `acedDefun()`:

Messages sent only if the application has registered an AutoLISP function	
Message	**Description**
kUnloadDwgMsg	Sent when the user quits a drawing session.
kInvkSubrMsg	Sent to invoke functions registered using acedDefun().
kEndMsg	Sent only when the END command is entered and there are changes that need to be saved (when dbmod != 0). kEndMsg is not sent for a NEW or OPEN, instead, kSaveMsg and kLoadDwgMsg are sent. For END, if dbmod = 0, then kQuitMsg is sent instead of kEndMsg.
kQuitMsg	Sent when AutoCAD quits (ends without saving) the drawing because a QUIT command was entered. The kQuitMsg can also be received with the END command, as noted above. If the END command is sent and dbmod = 0, then kQuitMsg is sent.
kSaveMsg	Sent when AutoCAD is saving the drawing because a SAVE, SAVEAS, NEW, or OPEN command is entered.
kCfgMsg	Sent when AutoCAD returns from the configuration program, and used only for a change to the display driver.

The next table lists the messages that an application receives if it has registered a service with ObjectARX.

Messages only received by applications that have registered a service	
Message	**Description**
kDependencyMsg	Sent when the ObjectARX application has registered an AcRxService object and the dependency count on that service changes from 0 to 1.
kNoDependencyMsg	Sent when the ObjectARX application has registered an AcRxService object and the dependency count on that service changes from 1 to 0.

The next table lists the messages that an application needs to respond to if it is using ActiveX Automation. See chapter 23, "COM, ActiveX Automation, and the Object Property Manager."

Messages only responded to by applications that use ActiveX Automation

Message	Description
kOleUnloadAppMsg	Sent to determine if the application can be unloaded (that is, none of its ActiveX objects or interfaces are being referenced by other applications).

See the *rxdefs.h* file where these enumeration constants are defined by the `AppMsgCode` type declaration.

You will need to decide which messages your ObjectARX application will respond to. The following table describes recommended actions upon receipt of a given message.

ObjectARX application reactions to AutoCAD messages

Message	Recommended Actions
kInitAppMsg	*Do* register services, classes, AcEd commands and reactors, and AcRxDynamicLinker reactors. Initialize application's system resources, such as devices and windows. Perform all one-time early initialization. AcRx, AcEd, and AcGe are all active. Store the value of the *pkt* parameter if you want to unlock and relock your application. *Don't* expect device drivers to be initialized, any user interface resources to be active, applications to be loaded in a particular order, AutoLISP to be present, or any databases to be open. Calls involving any of these assumptions will result in an error condition, sometimes fatal. AcDb and AcGi libraries are generally not yet active, although related AcRx and other structures are in place.
kUnloadAppMsg	*Do* perform final system resource cleanup. Anything started or created in kInitAppMsg should now be stopped or destroyed. *Don't* expect things to be any different from the description of kInitAppMsg. AutoCAD could be mostly dismantled by the time this call is made, except for the libraries listed as active in the kInitAppMsg *Do* description.

ObjectARX application reactions to AutoCAD messages (*continued*)

Message	Recommended Actions
kOleUnloadAppMsg	This message should be responded to only by applications using ActiveX Automation. *Do* respond with AcRx::kRetOK, if the application can be unloaded (none of its ActiveX objects or interfaces are being referenced by other applications). If it cannot be unloaded, respond with AcRx::kRetError.
kLoadDwgMsg	*Do* perform initialization relevant to the current drawing edit session. AcDb, AcGi, and the user interface API are all now active. Whether anything has been done to the drawing is not specified. All AutoCAD-supplied APIs are now active. You can perform AutoLISP function registration at this time, and initialize the user interface. Other operations to perform now include polling AutoCAD drivers and querying AcEditorReactor events if you want the earliest possible access to acdbHostApplicationServices()->workingDatabase(). *Don't* do anything you would not want to happen for every drawing edit session. Assume this message is sent more than once per program execution.
kUnloadDwgMsg	*Do* release or clean up everything started or registered in response to kLoadDwgMsg code. Release all AcDb reactors, excluding persistent reactors. *Don't* release system resources that are not tied to an edit session, or clean up AcRx classes, AcEd reactors, or commands; they remain valid *across* edit sessions.
kDependencyMsg	*Do* perform any actions that are necessary for your application when other applications become dependent on it, such as locking your application so that it cannot be unloaded.
kNoDependencyMsg	*Do* perform any actions that are necessary for your application when there are no longer any other applications dependent on yours, such as unlocking your application so that it can be unloaded by the user if desired.
kInvkSubrMsg	*Do* invoke the functions registered with acedDefun(). Determine the function by making a call to acedGetFuncode(). Return values with acedRetxxx(). *Don't* do much here except function invocation.
kPreQuitMsg	*Do* unload any dependencies (applications, DLLs, and so on) that your application controls to ensure that they are unloaded before your application.

ObjectARX application reactions to AutoCAD messages (*continued*)

Message	Recommended Actions
kEndMsg kCfgMsg kQuitMsg kSaveMsg	*Do* consider using the AcEditorReactor event callbacks as an alternative to responding to these messages. *Don't* respond to these messages if you're responding to the equivalent event callbacks made through AcEditorReactor.

Sequence of Events in an ObjectARX Application

The process of passing messages between AutoCAD and the ObjectARX application flows almost completely in one direction—from AutoCAD to the ObjectARX application. The following diagram shows a typical sequence for passing messages.

If an application is loaded when a drawing is already open, the `kInitAppMsg` and `kLoadDwgMsg` messages are sent in succession. When an ObjectARX application is unloaded while an edit session is in progress, the `kUnloadDwg` and `kUnloadApp` messages are sent in succession.

Implementing an Entry Point for AutoCAD

AutoCAD calls into the ObjectARX module through `acrxEntryPoint()`, which replaces the `main()` function of a C++ program. You are responsible for implementing the `acrxEntryPoint()` function, as described in this section.

The `acrxEntryPoint()` function serves as the entry point for AutoCAD (or other host programs) to communicate with an ObjectARX application. ObjectARX programs can in turn communicate with AutoCAD by returning status codes. All requests to invoke functions defined with `acedDefun()` are made by the `acrxEntryPoint()` function. If you define a new command with ObjectARX or with the `acedRegFunc()` function, AutoCAD immediately executes the function associated with the command (see "Loading an ObjectARX Application" on page 43).

The `acrxEntryPoint()` function has the following signature:

```
extern "C"
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt);
```

msg	Represents the message sent from the ObjectARX kernel to the application.
pkt	Holds packet data values.
AppRetCode	Contains the status code returned to AutoCAD.

Within the definition of the `acrxEntryPoint()` function, you write a switch statement or similar code to decipher messages from AutoCAD, perform appropriate actions related to each message, and return an integer status value.

WARNING! Using `kRetError` for the final return value from the `acrxEntryPoint()` function will cause your application to be unloaded, except for the messages `kOleUnloadAppMsg` and `kUnloadAppMsg`. In these cases, if `kRetError` is returned, the application will not be unloaded.

The following code shows the skeleton of a valid switch statement:

```
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
    switch(msg) {
        case AcRx::kInitAppMsg:
            break;
        case AcRx::kUnloadAppMsg:
            break;
        ...
        default:
            break;
    }
    return AcRx::kRetOK;
}
```

Initializing an ObjectARX Application

You must initialize any custom classes and commands that your application defines. This initialization can take place in either the `AcRx::kInitAppMsg` case of your `acrxEntryPoint()` function, or in a function called from that case.

To initialize an ObjectARX application

1. If you have defined a custom class, invoke its `rxInit()` function.

   Defining custom classes is discussed in detail in chapter 11, "Deriving a Custom ObjectARX Class."

2. If you have defined custom classes, call `acrxBuildClassHierarchy()` to rebuild the ObjectARX runtime class tree.

   For efficiency, call `acrxBuildClassHierarchy()` once after calling the `rxinit()` function for each of your custom classes.

3. Perform any other initialization that you need.

4. Register a service name.

   Registering a service name is suggested if other applications will depend upon your application. Registering a service name allows other applications to register depending on the service, and allows your application to check if it has any dependencies before unloading. Registering a service name for your application is also necessary if you are going to export symbolic functions from your application using the ObjectARX mechanism. You can use the function `acrxRegisterService()`, or use the `AcRxService` class. For more information on registering services, see the documentation on `AcRxService` in the *ObjectARX Reference*.

5. Register commands with the AutoCAD command mechanism.

Use `acedRegCmds->addCommand()` to make AutoCAD aware of the commands that your application defines. For more information, see "Registering New Commands" on page 40.

Preparing for Unloading

When your application is unloaded, you must clean up any custom classes or commands that your application has created. This should take place in the `AcRx::kUnloadAppMsg` case of your `acrxEntryPoint()` function, or in a function called from that case.

To unload an ObjectARX application

1. If you have created commands with the `acedRegCmds` macro or `acedDefun()`, remove them.

   Usually ObjectARX commands are removed by groups, using `acedRegCmds->removeGroup()`.

2. If you have created custom classes, remove them.

   Use the `deleteAcRxClass()` function to remove your custom classes from the AcRx runtime tree. Classes must be removed starting with the leaves of derived classes first, working up the class tree to parent classes.

3. Delete any objects added by the application.

   There is no way to tell AutoCAD to forget about `AcDbObject` instances that are currently resident in a database. However, when an application is unloaded, AutoCAD will automatically turn such objects into instances of `AcDbProxyObject` or `AcDbProxyEntity`.

4. Remove any reactors that have been attached to any `AcDbObject`, `AcDbDatabase`, `AcRxDynamicLinker`, or `AcEditor` object. (Persistent reactors on `AcDbObjects` are an exception; they will become proxy objects when the application is unloaded.)

5. If you have created a service name, remove it.

   You can use the `acrxServiceDictionary->remove()` function to remove any service that your application has registered. See the listing for `acrxServiceDictionary` in the *ObjectARX Reference*.

Example Application

The following example application implements functions that are called when the application is loaded and unloaded. Its initialization function adds two new commands to AutoCAD: CREATE and ITERATE. It also initializes the new class `AsdkMyClass` and adds it to the ObjectARX hierarchy with the `acrxBuildClassHierarchy()` function. (`AsdkMyClass` is described in "Example of a Custom Object Class" on page 338.)

```
// The initialization function called from the acrxEntryPoint()
// function during the kInitAppMsg case is used to add commands
// to the command stack and to add classes to the ACRX class
// hierarchy.
//
void
initApp()
{
    acedRegCmds->addCommand("ASDK_DICTIONARY_COMMANDS",
        "ASDK_CREATE", "CREATE", ACRX_CMD_MODAL,
        createDictionary);

    acedRegCmds->addCommand("ASDK_DICTIONARY_COMMANDS",
        "ASDK_ITERATE", "ITERATE", ACRX_CMD_MODAL,
        iterateDictionary);

    AsdkMyClass::rxInit();
    acrxBuildClassHierarchy();
}

// The cleanup function called from the acrxEntryPoint()
// function during the kUnloadAppMsg case removes this application's
// command set from the command stack and removes this application's
// custom classes from the ACRX runtime class hierarchy.
//
void
unloadApp()
{
    acedRegCmds->removeGroup("ASDK_DICTIONARY_COMMANDS");

    // Remove the AsdkMyClass class from the ACRX runtime
    // class hierarchy. If this is done while the database is
    // still active, it should cause all objects of class
    // AsdkMyClass to be turned into proxies.
    //
    deleteAcRxClass(AsdkMyClass::desc());
}
```

Registering New Commands

This section describes adding new commands using the AcEd command registration mechanism. For information on adding new commands using the functions `acedDefun()` and `acedRegFunc()`, see chapter 20, "ObjectARX Global Utility Functions." For information on adding new commands using the AutoLISP `defun` function, see the *AutoCAD Customization Guide*.

Command Stack

AutoCAD commands are stored in groups in the command stack, which is defined by the `AcEdCommandStack` class. One instance of the command stack is created per AutoCAD session. This stack consists of the custom commands that you have defined. The `acedRegCmds()` macro gives you access to the command stack.

When you add a command, you also assign it a group name. A good policy is to use your registered developer prefix for the group name to avoid name collisions with other commands. Command names within a given group must be unique, and group names must be unique. However, multiple applications can add a command of the same name, because the group name makes the commands unambiguous.

NOTE Autodesk supports a developer registration scheme to prevent namespace conflicts between different applications. Each registered developer chooses one or more registered developer symbols (RDS) to use exclusively. Registered developer symbols are one of the requirements of the "Built with ObjectARX" logo program. For more information, go online to *http://www.veritest.com/autodesk/main(f).htm.*

You usually add commands one at a time with the `AcEdCommandStack::addCommand()` function, and you remove commands by group with the `AcEdCommandStack::removeGroup()` function. You can also use the `AcEdCommandStack::removeCmd()` function to remove commands one at a time. As part of its cleanup before exiting, your application needs to remove any commands it registered.

The signature for the `addCommand()` function is

```
Acad::ErrorStatus
addCommand(
    const char* cmdGroupName,
    const char* cmdGlobalName,
    const char* cmdLocalName,
```

```
    Adesk::Int32 commandFlags,
    AcRxFunctionPtr functionAddr,
    AcEdUIContext *UIContext=NULL,
    int fcode=-1,
    HINSTANCE hResourceHandle=NULL);
```

cmdGroupName

ASCII representation of the group to add the command to. If the group doesn't exist, it is created before the command is added.

cmdGlobalName

ASCII representation of the command name to add. This name represents the global or untranslated name (see "Global versus Local Command Names" on page 42).

cmdLocalName

ASCII representation of the command name to add. This name represents the local or translated name.

commandFlags

Flags associated with the command. Possible values are ACRX_CMD_TRANSPARENT, ACRX_CMD_MODAL, ACRX_CMD_USEPICKSET, and ACRX_CMD_REDRAW (see "Transparent versus Modal Commands" on page 42).

functionAddr

Address of the function to be executed when this command is invoked by AutoCAD.

UiContext

Input pointer to `AcEdUIContext` callback class.

fcode

Input integer code assigned to the command.

NOTE It is strongly recommended that all command names be prefixed with your four-letter registered developer prefix to avoid possible conflicts with commands of the same name in other applications. For example, the name of a MOVE command for a developer with the prefix ASDK should be ASDKMOVE. Using your registered developer prefix is also recommended for group names.

The signature for the `removeCmd()` function is

```
virtual Acad::ErrorStatus
AcEdCommandStack::removeCmd
    (const char* cmdGroupName,
    const char* cmdGlobalName) = 0;
```

The signature for the `removeGroup()` function is

```
virtual Acad::ErrorStatus
AcEdCommandStack::removeGroup
    (const char* groupName);
```

Lookup Order

When a command is invoked, the command stack is searched by group name, then by command name within the group. In general, the first group registered will be the first one searched, but you cannot always predict what this order will be. Use the `AcEdCommandStack::popGroupToTop()` function to specify that a particular group should be searched first. At the user level, the Group option of the ARX command allows the user to specify which group to search first.

Global versus Local Command Names

When you add a command to AutoCAD, you need to specify both a global name that can be used in any language and a localized name that is a translated version of the command name to be used in a foreign-language version of AutoCAD. If you don't need to translate the command name into a local language, the same name can be used for both the global and local names.

Transparent versus Modal Commands

A command can be either transparent or modal. A transparent command can be invoked when the user is being prompted for input. A modal command can be invoked only when AutoCAD is posting the command prompt and no other commands or programs are currently active. The `commandFlags` argument to the `AcEdCommandStack::addCommand()` function specifies whether the new command is modal (ACRX_CMD_MODAL) or transparent (ACRX_CMD_TRANSPARENT). The `commandFlags` argument also specifies other options for the command. See `AcEdCommandStack` in the *ObjectARX Reference*. Transparent commands can be nested only one level (that is, the main command is invoked, which invokes one transparent command).

If you create multiple commands that operate on a common set of global objects, consider whether you should make them modal so that they won't interfere with each other. If such collisions are not a problem, making new commands transparent results in greater flexibility of use.

Loading an ObjectARX Application

You can load an ObjectARX application using any of the following methods:

- Provide the application with features that allow it to be demand loaded by AutoCAD. These features include application-specific entries in the Windows NT (or Windows® 95) system registry. See "Demand Loading" on page 45.
- Specify the application in the initial module file, *acad.rx*. This file contains ASCII text with the names of all programs AutoCAD should load when it is started. Each line in the file contains a program name (with the path if the file is not in a directory on the AutoCAD library search path). The *acad.rx* file must also be in a directory on the AutoCAD search path.
- Make an application load request from another ObjectARX application using `AcRxDynamicLinker::loadModule()`.
- Use the APPLOAD dialog box defined in the AutoCAD bonus program *loadapp.arx*.
- Use the `arxload()` function from AutoLISP.
- Use the `acedArxLoad()` function from ObjectARX.
- Enter the ARX command on the AutoCAD command line and use the Load option.

The Library Search Path

If you don't specify a search path, loading functions such as `arxload` search for the application in the directories specified by the AutoCAD library path. The AutoCAD library path includes the following directories in the order shown:

1. The current directory.
2. The directory that contains the current drawing file.
3. The directories specified by the support path (see the *AutoCAD Customization Guide*).
4. The directory that contains the AutoCAD program files.

Listing Loaded ObjectARX Applications

To see the names of all the ObjectARX programs currently loaded, use the Commands option of the ARX command. For more information, see "ARX Command" on page 53. The APPLOAD dialog box (defined in the AutoCAD bonus program *loadapp.arx*) also lists the names of the ObjectARX programs currently loaded.

Unloading an ObjectARX Application

You can unload an ObjectARX application with any of the following methods (if it is unlocked):

- Make an application unload request from another ObjectARX application using `AcRxDynamicLinker::unloadModule()`.
- Use the APPLOAD dialog box defined in the AutoCAD bonus program *loadapp.arx*. This file defines a user interface for the AutoLISP `arxload` and `arxunload` functions.
- Use the `arxunload` function from AutoLISP.
- Use the `acedArxUnload()` function from ObjectARX.
- Enter the ARX command on the AutoCAD command line and use the Unload option.

Unlocking Applications

By default, applications are locked and cannot be unloaded. To be classified as an "unloadable" application, the application must ensure that AutoCAD and other applications no longer refer to any objects or structures the application has defined. Before you make an application unloadable, be very careful that no client applications contain active pointers to any objects in your address space. For the list of cleanup operations an application must perform to be unloadable, see "Preparing for Unloading" on page 38.

If you want to make your application unloadable, you need to store the value of the `pkt` parameter sent with the `AcRx::kInitAppMsg`. The `pkt` parameter will be used by the `unlockApplication()` function. By default, an application is locked. If you unlock an application, it can be unloaded.

Use the following two functions to lock and unlock an application:

```
bool
AcRxDynamicLinker::lockApplication(void* pkt) const;

bool
AcRxDynamicLinker::unlockApplication(void* pkt) const;
```

The following function checks whether or not an application is locked:

```
bool
AcRxDynamicLinker::isApplicationLocked(const char* name)
                                       const;
```

Analogous global functions are also provided:

```
bool
acrxLockApplication(void* pkt);
```

```
bool
acrxUnlockApplication(void* pkt);

bool
acrxApplicationIsLocked(const char* modulename);
```

Demand Loading

Demand loading is a feature of AutoCAD that automatically attempts to load an ObjectARX application that is not resident in AutoCAD. ObjectARX applications can be designed for loading by AutoCAD under one or more of the following circumstances:

- When a drawing file that contains custom objects created by the absent application is read
- When a user or another application issues one of the absent application's commands
- When AutoCAD is started

NOTE Applications that implement demand loading on AutoCAD startup will be loaded *before* those listed in *acad.rx.*

Autodesk recommends developing ObjectARX applications that take advantage of AutoCAD's demand-loading feature because demand loading provides the following benefits:

- Limits the creation of proxy objects (see chapter 14, "Proxy Objects")
- Provides greater flexibility for loading ObjectARX applications
- Conserves memory by loading applications only when their functionality is required

For an application to be accessible for demand loading, application-specific information must be present in the Windows system registry. In addition, ObjectARX applications with more than one DLL may need a "controller" module that is responsible for loading all other components of the application. Finally, the DEMANDLOAD system variable must be set to the appropriate value for demand loading.

NOTE An ObjectARX application can be demand loaded from a path on the local machine, or by using an Internet address.

AutoCAD, the Windows System Registry, and ObjectARX Applications

AutoCAD uses the Windows system registry to maintain a wide range of application information, including information that uniquely identifies different AutoCAD releases, language versions, and products (such as AutoCAD Map®) that may be installed on any given computer. The registry information that identifies different versions of AutoCAD is of particular significance for ObjectARX developers. The installation program for an ObjectARX application must associate information about that ObjectARX application with information about the version(s) of AutoCAD with which it is supposed to run.

The AutoCAD installation program creates a unique time stamp key in the system registry immediately below the release number key (as well as adding the same installation ID to the executable itself). This key ensures that different versions of AutoCAD from the same release will be able to populate their own sections of the system registry. Within this key, values are stored for the location of AutoCAD files, the language version, and the product name, as illustrated in this example:

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\
    ACAD-1:409\
        ...
        AcadLocation:REG_SZ:f:\ACAD2000
        Language:REG_SZ:English
        ProductName:REG_SZ:AutoCAD Map R15.0
    ...
```

The installation program for an ObjectARX application must be able to locate the appropriate AutoCAD release key, as well as the appropriate language and product values.

The time stamp key is also used to identify the version of AutoCAD that is currently loaded (or the version that was most recently loaded). This identification is necessary, because the "current" version of AutoCAD resets the information in the global `HKEY_CLASSES_ROOT` section of the registry for its own use when it is loaded.

The `CurVer` value in the release key section of the registry is used to identify the current version, for example:

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\
    ...
    CurVer:REG_SZ:ACAD-1:409
```

When AutoCAD attempts to demand load an ObjectARX application, it looks in the section of the registry that belongs to the latest release of AutoCAD for information about the ObjectARX application. If it does not find the

ObjectARX information there, it checks the section for the previous release of AutoCAD, and so on in reverse order, until the information is found or the AutoCAD release information is exhausted.

Modification of the Registry at ObjectARX Application Installation

AutoCAD uses the Windows NT (or Windows 95) system registry to locate ObjectARX applications for demand loading. A part of the AutoCAD section of the registry is used for information about the location of ObjectARX applications' registry information.

The installation program for an ObjectARX application must create the specific keys and values in the system registry that are required for demand loading. Some of the required keys and values must be created in the AutoCAD section of the registry, and others must be created in the ObjectARX application's section of the registry.

If the ObjectARX application is designed to run with more than one version of AutoCAD (that is, different language versions or related products, such as AutoCAD Map), the installation program must add the appropriate information to the section of the registry for each version of AutoCAD.

The installation process for ObjectARX applications must therefore include:

- Verification that the sections of the system registry for the appropriate version of AutoCAD exist. (If the AutoCAD section of the registry does not exist, the user should be warned that a compatible version of AutoCAD has not been installed, and the installation should be aborted.)
- Creation of a specific set of keys and values for the application within the section(s) of the system registry for the appropriate version(s) of AutoCAD.
- Creation of a major key for the application itself, and population of that key with another set of specific keys and values.

See the *\objectarx\samples\polysamp\demandload* directory of the ObjectARX SDK for information about how the system registry is modified for demand loading the sample program *polysamp*.

The following two sections describe how an application's installation program should create the system registry information required for demand loading. A sample installation program is included in the *\objectarx\utils* directory of the ObjectARX SDK.

Creating AutoCAD Subkeys and Values

The ObjectARX application's installation program must be designed to manage a set of keys and values for that application within the section of system registry for each version of AutoCAD with which it is intended to run. The following example shows the layout of the keys and values in the section of the registry that must be created and maintained for the application:

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\releaseNum\
    ACAD-1:LocaleID\
        Applications\
            ApplicationName\
                LoadCtrls:REG_DWORD:acrxAppLoadReason
                RegPath:REG_SZ:RegistryPathWhereLoaderIsSpecified
```

The `releaseNum` and `ACAD-1:LocaleID` keys are created by the AutoCAD installation program.

The `ApplicationName` key must be the logical name of the application, which is used internally by AutoCAD to identify the program.

The `acrxAppLoadReason` value defines the conditions under which the application will be loaded, using one or more logical ORs of the following hex values listed with their associated meanings:

0x01	Load the application upon detection of proxy object.
0x02	Load the application upon AutoCAD startup.
0x04	Load the application upon invocation of a command.
0x08	Load the application upon request by the user or another application.
0x10	Do not load the application.

The `RegistryPathWhereLoaderIsSpecified` value must identify the registry path for the application's own section of the registry.

The ObjectARX API includes the `acrxRegisterApp()` function, which may be used in an ObjectARX application to enter information about the application into the AutoCAD section of the system registry. Typically, `acrxRegisterApp()` would enter this information the first time the application is loaded, and confirm the presence of that information on subsequent loads.

Creating ObjectARX Application Keys and Values

The ObjectARX application's installation program must be designed to manage the application's section of the system registry. This section of the registry must include keys and values identifying the main module of the application and the command set for the application.

The value in the `Loader` key must include the full path and file name of the module that AutoCAD should load first. The loader module is subsequently responsible for loading any other modules that make up the application.

The following example illustrates the layout and value types of the application section of the system registry:

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\
...
    RegistryPathWhereLoaderIsIdentified\
    Loader\Module:REG_SZ:DirPathFileName
    Name\DescriptiveName:REG_SZ:User Friendly App Name
    Commands\GlobalCommandName1:REG_SZ:LocalCommandName1
        GlobalCommandName2:REG_SZ:LocalCommandName2
        GlobalCommandName3:REG_SZ:LocalCommandName3
        GlobalCommandName4:REG_SZ:LocalCommandName4
        GlobalCommandName5:REG_SZ:LocalCommandName5
    Groups\
        GroupName:REG_SZ:GroupName
    ...
```

The `Module` value must be present but is not used except as a placeholder in the registry. Similarly, `User Friendly App Name` must be present, but is currently not used.

The value in the `Groups` key may be used to uniquely identify an ObjectARX application's command groups and therefore the commands as well.

Removing System Registry Information

It may be useful to remove ObjectARX application information from the system registry if an application is upgraded or removed. The ObjectARX API includes the function `acrxUnregisterApp()`, which is the counterpart of `acrxRegisterApp()`. It removes information about an application from the AutoCAD section of the system registry.

The DEMANDLOAD System Variable

The AutoCAD DEMANDLOAD system variable controls the demand loading options of ObjectARX applications.

By default the DEMANDLOAD system variable is set (when AutoCAD is installed) to enable demand loading of applications on command invocation or on proxy detection, when either option is specified in the system registry entry for an application. The setting of DEMANDLOAD does not affect demand loading on AutoCAD startup, or on request by a user or application when either of these options is specified in the system registry. (See "Creating AutoCAD Subkeys and Values" on page 48).

The legitimate values for the system variable may be used in combination. They are defined as follows:

0	Disables demand loading of all ObjectARX applications.
1	Enables demand loading of ObjectARX applications upon detection of proxy objects.
2	Enables demand loading of ObjectARX applications upon command invocation.
3	Enables demand loading for both proxy objects and command invocation (the default).

The DEMANDLOAD system variable allows the user to disable demand loading of all ObjectARX applications that have system registry settings specifying demand loading on command invocation, and proxy detection. It cannot cause an application to be demand loaded if the appropriate system registry settings do not exist.

Demand Loading on Detection of Custom Objects

When a DWG or DXF file containing custom objects is loaded, AutoCAD determines whether or not the associated application is loaded. If the application is not loaded, and the first bit of the system variable DEMANDLOAD is set, AutoCAD searches the Windows system registry for information about the application and its loader module. If AutoCAD finds the appropriate information in the system registry, it loads the application.

NOTE Demand loading on detection of custom classes will only work with classes that are derived from `AcDbObject`, either directly or indirectly.

As a hypothetical example, let's assume that AutoCAD reads a file created by the ObjectARX application *polysamp* (a product of PolySamp Inc.).

1. Upon reading the drawing file, AutoCAD encounters custom objects created with the application *polysamp*, and determines that the application is not loaded.
2. AutoCAD finds that the DEMANDLOAD system variable is set to enable demand loading of applications on proxy detection, so it searches the AutoCAD Applications section of the system registry for the `polysamp` key. Within this key, it finds the `LoadCtrls` value, which defines the conditions under which the application should be loaded, and the `RegPath` value, which

provides the full registry path for the *polysamp* module. This section of the registry would look something like this:

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\
    ACAD-1:409\
        Applications\PolyCAD\
            LoadCtrls:REG_DWORD:0xd
            RegPath:REG_SZ:
                \\HKEY_LOCAL_MACHINE\SOFTWARE\PolySampInc\polysamp
```

3 AutoCAD reads the `polysamp\Loader` key to determine the directory, path, and file name of the module to be loaded. This section of the directory would look something like this:

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\
    PolySampInc\polysamp\
        Loader\MODULE:REG_SZ:c:\polysampinc\arx\polyui.arx
        Name\PolySamp:REG_SZ:PolyCad
```

4 AutoCAD then attempts to load the ObjectARX module. If the module loads successfully, AutoCAD adds the application's handle to the list of application handles to be sent the `kLoadDwgMsg` message. AutoCAD then verifies that the application has been loaded properly, and verifies that the custom class is registered. If the application was loaded successfully, AutoCAD will continue to load the drawing file. If the ObjectARX module cannot be loaded, or if there still isn't a class implementation available, custom objects are treated as proxies and the load continues.

Demand Loading on Command

AutoCAD will attempt to load the appropriate ObjectARX application if the user invokes a command that is not registered with AutoCAD.

To support demand loading on command invocation, the ObjectARX application's installation program must create the appropriate keys and values in the system registry for the application's commands. The application's `Commands` section of the system registry should contain command information like this:

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\
    Autodesk\ ...
...
    PolySampInc\polysamp\
        Loader\MODULE:REG_SZ:c:\polysampinc\arx\polyui.arx
        Name\PolySamp:REG_SZ:PolyCad
        Commands\
            ASDKPOLY:REG_SZ:ASDKPOLY
            ASDKDRAGPOLY:REG_SZ:ASDKDRAGPOLY
            ASDKPOLYEDIT:REG_SZ:ASDKPOLYEDIT
        Groups\
            ASDK:REG_SZ:ASDK
...
```

In this example, the developer's registered developer prefix (ASDK) is used as the prefix for all commands to ensure that there will be no possible conflict with commands of the same name in other applications.

The ObjectARX application must also include the appropriate calls to the `acedRegCmds` macro for demand loading on command to work.

Demand Loading on AutoCAD Startup

Demand loading of an ObjectARX application on AutoCAD startup can be specified by using `0x02` (or you can perform an OR `0x02` with another legitimate value) with the `LoadCtrls` value in the system registry, as shown here.

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\
    ACAD-1:409\
        Applications\PolyCAD\
            LoadCtrls:REG_DWORD:0x02
            RegPath:REG_SZ:
```

Managing Applications with the System Registry

Once system registry information has been created for demand loading, that same information can be used by a set of ObjectARX functions to load, unload, and monitor the presence of ObjectARX applications independent of the demand-loading feature. The **`AppName`** argument used by the first two of these functions is the logical application name.

The following ObjectARX functions can be used with registered application names:

```
bool acrxLoadApp ("AppName")
```

This function takes a single argument, which represents the case-insensitive logical name of the application to be loaded. The function returns 0 if the load failed, or 1 if the load succeeds.

```
bool acrxUnloadApp ("AppName")
```

This function takes a single argument, which represents the case-insensitive logical name of the application that was previously loaded. The function returns 0 if the unload fails, or 1 if it succeeds.

```
void *acrxLoadedApps ()
```

This function returns an array of strings as a `void *`, containing the logical application name of each application that is currently loaded. The function returns `NULL` if no applications are loaded. It is the caller's responsibility to release the space allocated for the returned strings.

ARX Command

The following sections describe the ARX command and its options. The initial prompt is as follows:

?/Load/Unload/Commands/Options: *Enter an option or press* ENTER

?—List Applications

Lists the currently loaded ARX applications.

Load

Loads the *.arx* file that you specify in the standard file dialog box. If FILEDIA is set to 0, a dialog box is not displayed, and you enter the name of the file to load in response to the following prompt:

Runtime extension file: *Enter a name*

Unload

Unloads the specified ARX program. Some applications cannot be unloaded. See "Unloading an ObjectARX Application" on page 44 for a description of how the programmer decides whether a program can be unloaded by the user with this command.

Commands

Displays all command names in all command groups registered from ARX programs.

Options

Presents developer-related ARX application options.

Options (Group/CLasses/Services): *Enter an option*

- Group
  Moves the specified group of commands registered from ARX applications to be the first group searched when resolving the names of AutoCAD commands. Other registered groups, if there are any, are subsequently searched, in the same order as before the ARX command was executed.

  Command Group Name: *Enter the command group name*

The search order is important only when a command name is listed in multiple groups. This mechanism allows different ARX applications to define the same command names in their own separate command groups. ARX applications that define command groups should publish the group name in their documentation.

Group is not intended to be selected by the user directly. The user specifies which group is searched first by interacting with a script that executes the ARX command with the Group option. This capability is usually embedded in key menu item scripts. The user selects a menu item from the script. The key menu item script executes the Group option to establish which group is searched first, giving commands of the same name (but probably different functionality) from one application precedence over commands from another.

For example, applications called ABC Construction and XYZ Interiors define command groups ABC and XYZ, respectively. Most of ABC Construction's commands are named with construction terminology, while most of XYZ Interiors' commands are named with interior decorating terminology, but both applications define commands named INVENTORY and ORDERS. When working on the construction aspects of a drawing, the user chooses a menu item defined by ABC Construction, and the following script runs:

```
ARX
Group
ABC
```

The script pops the ABC Construction command set to give it top priority and to resolve INVENTORY to the ABC Construction version of the command. Later, when an interior designer is working on the drawing with the same set of applications loaded, selecting a key icon ensures that the XYZ Interiors commands have precedence.

NOTE Command groups are not related to commands defined in AutoLISP or defined by a call to `acedDefun()` by ObjectARX applications. The software mechanism that defines command groups is described in "Lookup Order" on page 42.

- Classes
  Displays a class hierarchy of C++ classes derived from objects registered in the system, whether registered by AutoCAD or by an ARX program.
- Services
  Lists the names of all services registered by AutoCAD and by loaded ARX programs.

Running ObjectARX Applications from AutoLISP

An ObjectARX application can define a set of functions known to AutoLISP as external functions, by using `acedDefun()`. After the application is loaded, you can invoke an external function exactly as you can invoke a built-in or user-defined AutoLISP function. AutoLISP variables can be passed as arguments to the external function, and the external function can return a result. The external function can also prompt the user to enter data, either from the keyboard or by specifying points or objects with the pointing device, and the external function can set Windows or AutoCAD platform-independent help.

The external function can be invoked by an AutoLISP function, as well as interactively. ObjectARX applications cannot call AutoLISP functions. An ObjectARX application can retrieve and set the value of AutoLISP symbols (the symbol's data type must be recognizable to a C++ program).

An ObjectARX application can define a new AutoCAD command with the same `c:xxx` convention as AutoLISP. You invoke the external function by entering its name at the Command prompt, with no parentheses.

Defining an external function replaces any previous definition of the same name. If two ObjectARX applications define functions with the same name, the function in the first application to be loaded is lost; if you unload the second application, you cannot call the duplicate function.

Error Handling

The examples in this guide have omitted necessary error checking to simplify the code. However, you'll always want to check return status and take appropriate action. The following example shows appropriate use of error checking for several examples shown first in chapter 2, "Database Primer."

```
Acad::ErrorStatus
createCircle(AcDbObjectId& circleId)
{
    circleId = AcDbObjectId::kNull;

    AcGePoint3d center(9.0, 3.0, 0.0);
    AcGeVector3d normal(0.0, 0.0, 1.0);
    AcDbCircle *pCirc = new AcDbCircle(center, normal, 2.0);
```

```
if (pCirc == NULL)
    return Acad::eOutOfMemory;

AcDbBlockTable *pBlockTable;
Acad::ErrorStatus es =
    acdbHostApplicationServices()->workingDatabase()->
        getSymbolTable(pBlockTable, AcDb::kForRead);
if (es != Acad::eOk) {
    delete pCirc;
    return es;
}

AcDbBlockTableRecord *pBlockTableRecord;
es = pBlockTable->getAt(ACDB_MODEL_SPACE,
    pBlockTableRecord, AcDb::kForWrite);
if (es != Acad::eOk) {
    Acad::ErrorStatus es2 = pBlockTable->close();
    if (es2 != Acad::eOk) {
        acrx_abort("\nApp X failed to close Block"
            " Table. Error: %d",
            acadErrorStatusText(es2));
    }
    delete pCirc;
    return es;
}

es = pBlockTable->close();
if (es != Acad::eOk) {
    acrx_abort("\nApp X failed to close Block Table."
        " Error: %d", acadErrorStatusText(es));
}

es = pBlockTableRecord->appendAcDbEntity(circleId,
    pCirc);
if (es != Acad::eOk) {
    Acad::ErrorStatus es2 = pBlockTableRecord->close();
    if (es2 != Acad::eOk) {
        acrx_abort("\nApp X failed to close"
            " Model Space Block Record. Error: %s",
            acadErrorStatusText(es2));
    }
    delete pCirc;
    return es;
}

es = pBlockTableRecord->close();
if (es != Acad::eOk) {
    acrx_abort("\nApp X failed to close"
        " Model Space Block Record. Error: %d",
        acadErrorStatusText(es));
}
```

```
    es = pCirc->close();
    if (es != Acad::eOk) {
        acrx_abort("\nApp X failed to"
            " close circle entity. Error: %d",
            acadErrorStatusText(es));
    }
    return es;
}

Acad::ErrorStatus
createNewLayer()
{
    AcDbLayerTableRecord *pLayerTableRecord
        = new AcDbLayerTableRecord;

    if (pLayerTableRecord == NULL)
        return Acad::eOutOfMemory;

    Acad::ErrorStatus es
        = pLayerTableRecord->setName("ASDK_MYLAYER");
    if (es != Acad::eOk) {
        delete pLayerTableRecord;
        return es;
    }

    AcDbLayerTable *pLayerTable;
    es = acdbHostApplicationServices()->workingDatabase()->
        getSymbolTable(pLayerTable, AcDb::kForWrite);
    if (es != Acad::eOk) {
        delete pLayerTableRecord;
        return es;
    }

    // The linetype object ID default is 0, which is
    // not a valid ID.  Therefore, it must be set to a
    // valid ID, the CONTINUOUS linetype.
    // Other data members have valid defaults, so
    // they can be left alone.
    //
    AcDbLinetypeTable *pLinetypeTbl;
    es = acdbHostApplicationServices()->workingDatabase()->
        getSymbolTable(pLinetypeTbl, AcDb::kForRead);
    if (es != Acad::eOk) {
        delete pLayerTableRecord;
        es = pLayerTable->close();
        if (es != Acad::eOk) {
            acrx_abort("\nApp X failed to close Layer"
                " Table. Error: %d",
                acadErrorStatusText(es));
        }
        return es;
    }
```

```
    AcDbObjectId ltypeObjId;
    es = pLinetypeTbl->getAt("CONTINUOUS", ltypeObjId);
    if (es != Acad::eOk) {
        delete pLayerTableRecord;
        es = pLayerTable->close();
        if (es != Acad::eOk) {
            acrx_abort("\nApp X failed to close Layer"
                " Table. Error: %d",
                acadErrorStatusText(es));
        }
        return es;
    }
    pLayerTableRecord->setLinetypeObjectId(ltypeObjId);
    es = pLayerTable->add(pLayerTableRecord);
    if (es != Acad::eOk) {
        Acad::ErrorStatus es2 = pLayerTable->close();
        if (es2 != Acad::eOk) {
            acrx_abort("\nApp X failed to close Layer"
                " Table. Error: %d",
                acadErrorStatusText(es2));
        }
        delete pLayerTableRecord;
        return es;
    }

    es = pLayerTable->close();
    if (es != Acad::eOk) {
        acrx_abort("\nApp X failed to close Layer"
            " Table. Error: %d",
            acadErrorStatusText(es));
    }

    es = pLayerTableRecord->close();
    if (es != Acad::eOk) {
        acrx_abort("\nApp X failed to close Layer"
            " Table Record. Error: %d",
            acadErrorStatusText(es));
    }
    return es;
}
```

Database Operations

In This Chapter

4

- Initial Database
- Creating and Populating a Database
- Saving a Database
- The wblock Operation
- Inserting a Database
- Setting Current Database Values
- Example of Database Operations
- Long Transactions
- External References
- Indexes and Filters
- Drawing Summary Information
- Last Saved by Autodesk Software

This chapter describes basic database protocol including how to create a database, how to read in a drawing file, and how to save the database. The `wblock` and `insert` operations are also described here.

For more detailed information on the `deepClone` and `wblock` operations, see chapter 18, "Deep Cloning."

Initial Database

When an AutoCAD session begins, the database contains the following elements:

- *A set of nine symbol tables.*

  Block table (`AcDbBlockTable`)
  Dimension style table (`AcDbDimStyleTable`)
  Layer table (`AcDbLayerTable`)
  Linetype table (`AcDbLinetypeTable`)
  Registered applications table (`AcDbRegAppTable`)
  Text style table (`AcDbTextStyleTable`)
  User Coordinate System table (`AcDbUCSTable`)
  Viewport table (`AcDbViewportTable`)
  View table (`AcDbViewTable`)

  Some of the symbol tables already contain one or more records. The layer table in a pristine database contains one record, layer 0. The block table initially contains three records: *MODEL_SPACE, *PAPER_SPACE, and *PAPER_SPACE0. The linetype table always has CONTINUOUS, BY_LAYER, and BY_BLOCK linetype table records. The registered applications table always has an ACAD table record. The text style table always has a STANDARD table record.
- *A named object dictionary.* When a database is created, this dictionary already contains the two database dictionaries: the GROUP dictionary and the MLINE style dictionary. Within the MLINE style dictionary, the STANDARD style is always present.
- *A fixed set of header variables.* (These are not database objects.)

Creating and Populating a Database

Use `new` to create a database and `delete` to destroy one. The `AcDbDatabase` constructor has one argument with a default value of `Adesk::kTrue`. If this argument is `Adesk::kTrue`, then the database is populated with the standard database objects, described in "Initial Database." If the argument is `Adesk::kFalse`, then an empty database is created and can be populated by reading in a drawing file.

Use the following function to read in a drawing file:

```
AcadErrorStatus
AcDbDatabase::readDwgFile(char* fileName);
```

If you receive any of the following error codes, you probably want to recover the drawing with the standard AutoCAD recover mechanism provided by the user interface:

```
kDwgNeedsRecovery
kDwgCRCDoesNotMatch
kDwgSentinelDoesNotMatch
kDwgObjectImproperlyRead
```

WARNING! Never delete the database returned by the `acdbHostApplicationServices()->workingDatabase()` function.

Saving a Database

To save a database, use the `AcDbDatabase::saveAs()` function:

```
Acad::ErrorStatus
AcDbDatabase::saveAs(char* fileName);
```

The file name can be a path to a local file, or an Internet address.

Setting the Default File Format

ObjectARX provides the ability to specify the default file format for the SAVEAS, SAVE, and QSAVE commands. (The AUTOSAVE command always saves drawings in the AutoCAD 2000 drawing file format.)

The class `AcApDocument` contains an enumeration that defines the format used when saving a drawing to a file. Its values are shown in the following table:

Save format

Name	Usage (file extension)
kR12_dxf	AutoCAD Release 12/LT2 DXF (*.dxf)
kR13_dwg	AutoCAD Release 13/LT95 Drawing (*.dwg)
kR13_dxf	AutoCAD Release 13/LT95 DXF (*.dxf)
kR14_dwg	AutoCAD Release 14/LT97 Drawing (*.dwg)
kR14_dxf	AutoCAD Release 14/LT97 DXF (*.dxf)

Save format (*continued*)	
Name	**Usage (file extension)**
kR15_dwg	AutoCAD 2000 Drawing (*.dwg)
kR15_dxf	AutoCAD 2000 DXF (*.dxf)
kR15_Template	AutoCAD 2000 Drawing Template File (*.dwt)
kNative	Current DWG version is AutoCAD 2000
kUnknown	Invalid format

The `AcApDocument::formatForSave()` function returns the current save format being used by the SAVEAS, SAVE, and QSAVE commands:

```
AcApDocument::SaveFormat
formatForSave();
```

The value returned may be either the session-wide default setting, or a different setting that the user has selected for this document. If it is an override for this document, it will not persist across sessions.

The `AcApDocmanager::setDefaultFormatForSave()` function uses one of the `SaveFormat` values to set the file format to use when saving a drawing with the SAVEAS, SAVE, and QSAVE commands. This sets the session-wide default, which the user may choose to temporarily override for an individual document:

```
Acad::ErrorStatus
setDefaultFormatForSave(
    AcApDocument::SaveFormat format);
```

These functions only directly report on or set the file format for interactive commands entered by the user. If you want your application to use the current save format, every time you wish to save the database, you will first need to call `formatForSave()`, and then use the returned `SaveFormat` value to determine which function to call. For example, if `formatForSave()` returned `kR14_dxf`, you would call `acdbDxfOutAsR14()` to write the database as a Release 14 DXF file.

Be sure to take the following into account:

- Either you or your user may set a persistent session-wide default format for save that will be honored by all save commands except AUTOSAVE.
- Only the user can temporarily (not persistently between sessions) override this setting for a particular document.

- The `formatForSave()` method returns the format in which the user wishes an individual document to be saved; this will be either the session-wide default or the temporary override, as appropriate.

Global Save Functions

ObjectARX also contains two global functions for saving drawings:

```
Acad::ErrorStatus
acdbSaveAsR13(
    AcDbDatabase* pDb,
    const char* fileName);
```

```
Acad::ErrorStatus
acdbSaveAsR14(
    AcDbDatabase* pDb,
    const char* fileName);
```

Both functions accept a database pointer and a filename, and write out the drawing in AutoCAD Release 13 or Release 14 DWG format, respectively.

The wblock Operation

The `AcDbDatabase` class contains an overloaded `wblock()` function with three forms that correspond to the options of the AutoCAD WBLOCK command.

Creating a New Database from an Existing Database

The following function is the equivalent of the WBLOCK* command:

```
Acad::ErrorStatus
AcDbDatabase::wblock(AcDbDatabase*& newDb);
```

This function creates a new database from the invoked database ("`this`"). Any unreferenced symbols in the input database are omitted in the new database (which makes the new database potentially cleaner and smaller than the original). However, it does not take care of copying application-defined objects whose ownership is rooted in the named object dictionary. You need to transfer application data from the source database to the target database using the `AcEditorReactor` notification functions.

Creating a New Database with Entities

The other two forms of the `AcDbDatabase::wblock()` function create a new database whose model space block table record contains the specified entities from the input database. The first form of this function copies the entities from a named block table record. The second form of the function copies an array of entities.

Copying a Named Block

The following function is equivalent to invoking the WBLOCK command with the name of a block definition:

```
Acad::ErrorStatus
AcDbDatabase::wblock(AcDbDatabase*& newDb,
                     AcDbObjectId recordId);
```

The `recordId` argument represents a block table record in the input database. The entities in this block table record are copied into the new database's model-space block table record. The insert base of the new database is the block table record's origin.

Copying an Array of Entities

The following function is equivalent to invoking the WBLOCK command and then using the option to select specific objects and specify an insertion base point:

```
Acad::ErrorStatus
AcDbDatabase::wblock(AcDbDatabase*& newDb,
                     const AcDbObjectIdArray& idArray,
                     const AcGePoint3d* point);
```

This function creates a new database that includes the entities specified in the `idArray` argument. The entities, which can be in the model space or paper space block table records of the input database, are placed in the model space of the new database. Also included in the new database are the objects owned by or referred to by those entities, as well as the owners of those objects. The specified point is the origin point, in world coordinates, for the new drawing (that is, it is the insert base point in the model space of the new database).

Inserting a Database

The `AcDbDatabase::insert()` functions copy one database into the database that the member function is invoked on. AutoCAD merges the objects that it defines, such as the MLINE style and GROUP dictionaries; however, it does not take care of copying application-defined objects whose ownership is rooted in the named object dictionary. You need to transfer application data from the source database to the target database using the `AcEditorReactor` notification functions.

NOTE The `insert()` functions perform deep cloning, as described in chapter 18, "Deep Cloning."

If conflicts arise when the source and target databases are being merged (for example, if both databases have the same linetype name), AutoCAD uses the version in the target database.

The following function is equivalent to a standard drawing INSERT command:

```
Acad::ErrorStatus
AcDbDatabase::insert(AcDbObjectId& blockId,
                     const char* pBlockName,
                     AcDbDatabase* pDb);
```

This function copies the entities from the model space of the input database (`pDb`) into the specified block table record (`pBlockName`) and returns the block ID of the new block table record (`blockId`). The application must then create the reference to the block table record and add it to the database.

The following function is equivalent to an AutoCAD INSERT* command:

```
Acad::ErrorStatus
AcDbDatabase::insert(const AcGeMatrix3d& xform,
                     AcDbDatabase* pDb);
```

This function copies the entities from the model space of the input database (`pDb`) and puts them into the current space of the new database (paper space or model space), applying the specified transformation (`xform`) to the entities.

Setting Current Database Values

If a data property such as color or linetype is not specified for an entity, the database's current value for that data is used. The following sections outline the functions used to specify the current data values associated with the database.

Database Color Value

If a color is not specified for an entity, the database's current color value, stored in the CECOLOR system variable, is used. The following functions set and retrieve the current color value in the database:

```
Acad::ErrorStatus
AcDbDatabase::setCecolor(const AcCmColor& color);

AcCmColor AcDbDatabase::cecolor() const;
```

Database Linetype Value

The following functions set and retrieve the current linetype value in the database:

```
Acad::ErrorStatus
AcDbDatabase::setCeltype(AcDbObjectId);

AcDbObjectId AcDbDatabase::celtype() const;
```

Database Linetype Scale Value

The database has three linetype scale settings:

- A linetype scale setting for the current entity, stored in the CELTSCALE system variable.
- A linetype scale setting for the current drawing, stored in the LTSCALE system variable.
- A flag that indicates whether to apply linetype scaling to the space the entity resides in or to the entity's appearance in paper space. This setting is stored in the PSLTSCALE system variable.

The global LTSCALE and PSLTSCALE settings are used when a drawing is regenerated (see chapter 6, "Entities"). Use the following functions to set and inquire these values:

```
Acad::ErrorStatus
AcDbDatabase::setLtscale(double);

double AcDbDatabase::ltScale() const;


Acad::ErrorStatus
AcDbDatabase::setCeltscale(double);

double AcDbDatabase::celtscale() const;


Acad::ErrorStatus
AcDbDatabase::setPsltscale(Adesk::Boolean)

Adesk::Boolean AcDbDatabase::psltscale() const;
```

Database Layer Value

The following functions set and retrieve the current layer value in the database:

```
Acad::ErrorStatus
AcDbDatabase::setClayer(AcDbObjectId);

AcDbObjectId AcDbDatabase::clayer() const;
```

Example of Database Operations

The following example shows the `createDwg()` routine, which creates a new database, obtains the model space block table record, and creates two circles that are added to model space. It uses the `AcDbDatabase::saveAs()` function to save the drawing. The second routine, `readDwg()`, reads in the saved drawing, opens the model space block table record, and iterates through it, printing the class names of the entities it contains.

```
void
createDwg()
{
    AcDbDatabase *pDb = new AcDbDatabase();

    AcDbBlockTable *pBtbl;
    pDb->getSymbolTable(pBtbl, AcDb::kForRead);
```

```
    AcDbBlockTableRecord *pBtblRcd;
    pBtbl->getAt(ACDB_MODEL_SPACE, pBtblRcd,
        AcDb::kForWrite);
    pBtbl->close();

    AcDbCircle *pCir1 = new AcDbCircle(AcGePoint3d(1,1,1),
                                       AcGeVector3d(0,0,1),
                                       1.0),
               *pCir2 = new AcDbCircle(AcGePoint3d(4,4,4),
                                       AcGeVector3d(0,0,1),
                                       2.0);
    pBtblRcd->appendAcDbEntity(pCir1);
    pCir1->close();

    pBtblRcd->appendAcDbEntity(pCir2);
    pCir2->close();
    pBtblRcd->close();

    // AcDbDatabase::saveAs() does not automatically
    // append a DWG file extension, so it
    // must be specified.
    //
    pDb->saveAs("test1.dwg");
    delete pDb;
}

void
readDwg()
{
    // Set constructor parameter to kFalse so that the
    // database will be constructed empty.  This way only
    // what is read in will be in the database.
    //
    AcDbDatabase *pDb = new AcDbDatabase(Adesk::kFalse);

    // The AcDbDatabase::readDwgFile() function
    // automatically appends a DWG extension if it is not
    // specified in the filename parameter.
    //
    pDb->readDwgFile("test1.dwg");

    // Open the model space block table record.
    //
    AcDbBlockTable *pBlkTbl;
    pDb->getSymbolTable(pBlkTbl, AcDb::kForRead);

    AcDbBlockTableRecord *pBlkTblRcd;
    pBlkTbl->getAt(ACDB_MODEL_SPACE, pBlkTblRcd,
        AcDb::kForRead);
    pBlkTbl->close();

    AcDbBlockTableRecordIterator *pBlkTblRcdItr;
    pBlkTblRcd->newIterator(pBlkTblRcdItr);
```

```
    AcDbEntity *pEnt;
    for (pBlkTblRcdItr->start(); !pBlkTblRcdItr->done();
        pBlkTblRcdItr->step())
    {
        pBlkTblRcdItr->getEntity(pEnt,
            AcDb::kForRead);
        acutPrintf("classname: %s\n",
            (pEnt->isA())->name());
        pEnt->close();
    }
    pBlkTblRcd->close();

    delete pBlkTblRcdItr;
    delete pDb;
}
```

Long Transactions

Long Transactions are used to support the AutoCAD Reference Editing feature and are very useful for ObjectARX applications. These classes and functions provide a scheme for applications to check out entities for editing and check them back in to their original location. This operation replaces the original objects with the edited ones. There are three types of long transaction check out:

- From a normal block within the same drawing
- From an external reference (xref) of the drawing
- From an unrelated, temporary database

Class and Function Overview

The main classes and functions are

- `AcDbLongTransaction` class
- `AcDbLongTransWorkSetIterator` class
- `AcApLongTransactionReactor` class
- `AcApLongTransactionManager` class
- `AcDbDatabase::wblockCloneObjects()` function

AcDbLongTransaction Class

`AcDbLongTransaction` is the class that contains the information needed to track a long transaction. The `AcDbLongTransactionManager` class takes the responsibility for creating and appending `AcDbLongTransaction` objects to the database. It then returns the `AcDbObjectId` of the `AcDbLongTransaction`

object. Like all other database-resident objects, its destruction is handled by the database.

NOTE The `AcDbLongTransaction` objects are added to a database while they are active and are erased once the transaction has completed. They are not stored in DWG or DXF files, and therefore are not persistent.

AcDbLongTransWorkSetIterator Class

`AcDbLongTransWorkSetIterator` provides read-only access to the objects in the work set. During construction in `AcDbLongTransaction::newWorkSetIterator()`, it can be set up to include only the active work set, or include objects added to the work set because they are referenced by objects in the work set (secondary objects). It can also handle objects removed from the work set, either by `AcDbLongTransaction::removeFromWorkSet()`, or by being erased.

AcApLongTransactionReactor Class

`AcApLongTransactionReactor` provides notification specific to long transaction operations. It is designed to be used in conjunction with the deep clone notifications that will also be sent, but will vary depending upon which type of check out/in is being executed. To connect these notifications with the deep clone notifications, the `AcDbIdMapping` object used for cloning can be retrieved by calling the `AcDbLongTransaction::activeIdMap()` function.

AcApLongTransactionManager Class

`AcApLongTransactionManager` is the manager for starting and controlling long transactions. There is only one for each AutoCAD session, and it is accessed via a pointer returned by the `acapLongTransactionManager` object.

AcDbDatabase::wblockCloneObjects() Function

The `wblockCloneObjects()` function is a member of `AcDbDatase`. It will deep clone objects from one database to another and follow hard references so that all dependent objects are also cloned. The behavior of symbol table records, when duplicates are found, is determined by the type parameter. The

following chart shows the relationship between a symbol table type (`enum DuplicateRecordCloning`) and a deep clone type (`enum DeepCloneType`).

Relationship between DeepCloneTypes and DuplicateRecordCloning for different commands and functions

Command or API Function	DeepCloneType	DuplicateRecordCloning
COPY	kDcCopy	kDrcNotApplicable
EXPLODE	kDcExplode	kDrcNotApplicable
BLOCK	kDcBlock	kDrcNotApplicable
INSERT/BIND	kDcXrefInsert	kDrcIgnore
XRESOLVE	kDcSymTableMerge	kDrcXrefMangleName
INSERT	kDcInsert	kDrcIgnore
insert()	kDcInsertCopy	kDrcIgnore
WBLOCK	kDcWblock	kDrcNotApplicable
deepCloneObjects()	kDcObjects	kDrcNotApplicable
wblockObjects()	kDcObjects	kDrcIgnore
wblockObjects()	kDcObjects	kDrcReplace
wblockObjects()	kDcObjects	kDrcMangleName
wblockObjects()	kDcObjects	kDrcUnmangleName

Long Transaction Example

This simple example shows how to check out entities from another database, modify them in the current database, and then check them back in to the original database. The calls that are part of the long transaction process are indicated in bold print.

```
void
refEditApiExample()
{
    AcDbObjectId transId;
    AcDbDatabase* pDb;
    char *fname;
    struct resbuf *rb;
```

```
// Get a dwg file from the user.
//
rb = acutNewRb(RTSTR);
acedGetFileD("Pick a drawing", NULL, "dwg", 0, rb);
fname = (char*)acad_malloc(strlen(rb->resval.rstring) + 1);
strcpy(fname, rb->resval.rstring);
acutRelRb(rb);

// Open the dwg file.
//
pDb = new AcDbDatabase(Adesk::kFalse);
pDb->readDwgFile(fname);

// Get the block table and then the model space record.
//
AcDbBlockTable *pBlockTable;
pDb->getSymbolTable(pBlockTable, AcDb::kForRead);
AcDbBlockTableRecord *pOtherMsBtr;
pBlockTable->getAt(ACDB_MODEL_SPACE, pOtherMsBtr,
      AcDb::kForRead);
pBlockTable->close();


// Create an iterator
//
AcDbBlockTableRecordIterator *pIter;
pOtherMsBtr->newIterator(pIter);


// Set up an object ID array.
//
AcDbObjectIdArray objIdArray;


// Iterate over the model space BTR. Look specifically
// for Lines and append their object ID to the array.
//
for (pIter->start(); !pIter->done(); pIter->step()) {
    AcDbEntity *pEntity;
    pIter->getEntity(pEntity, AcDb::kForRead);
    // Look for only AcDbLine objects and add them to the
    // objectId array.
    //
    if (pEntity->isKindOf(AcDbLine::desc())) {
        objIdArray.append(pEntity->objectId());
    }
    pEntity->close();
}
delete pIter;
pOtherMsBtr->close();
```

```
// Now get the current database and the object ID for the
// current database's model space BTR.
//
AcDbBlockTable *pThisBlockTable;
acdbHostApplicationServices()->workingDatabase()
    ->getSymbolTable(pThisBlockTable, AcDb::kForRead);
AcDbBlockTableRecord *pThisMsBtr;
pThisBlockTable->getAt(ACDB_MODEL_SPACE, pThisMsBtr,
      AcDb::kForWrite);

pThisBlockTable->close();

AcDbObjectId id = pThisMsBtr->objectId();
pThisMsBtr->close();


// Create the long transaction. This will check all the entities
// out of the external database.
acapLongTransactionManagerPtr()->checkOut(transId,
      objIdArray, id);

// Now modify the color of these entities.
//
int colorIndex;
acedGetInt("\nEnter color number to change entities to: ",
      &colorIndex);

AcDbObject* pObj;
if (acdbOpenObject(pObj, transId, AcDb::kForRead) == Acad::eOk)
{
     // Get a pointer to the transaction
     //
     AcDbLongTransaction* pLongTrans =
         AcDbLongTransaction::cast(pObj);

     if (pLongTrans != NULL) {
         // Get a work set iterator
         //
         AcDbLongTransWorkSetIterator* pWorkSetIter =
            pLongTrans->newWorkSetIterator();

         // Iterate over the entities in the work set
         // and change the color.
         for (pWorkSetIter->start(); !pWorkSetIter->done();
              pWorkSetIter->step()) {

             AcDbEntity *pEntity;
          acdbOpenAcDbEntity(pEntity, pWorkSetIter->objectId(),
                 AcDb::kForWrite);
             pEntity->setColorIndex(colorIndex);
             pEntity->close();
         }
         delete pWorkSetIter;
     }
     pObj->close();
}
```

```
    // Pause just to see the change.
    //
    char str[132];
    acedGetString(0, "\nNote the new colors. Press return to \
        check the objects back in to the original database", str);

    // Check the entities back in to the orginal database.
    //
    acapLongTransactionManagerPtr()->checkIn(transId);
    // Save the original database, since we made changes
    //
    pDb->saveAs(fname);
    // Close and Delete the database.
    //
    delete pDb;
    pDb = NULL;
    acad_free(fname);
}
```

External References

External references (xrefs) can be created and manipulated through several global functions. These global functions mimic the AutoCAD XREF command capabilities. The functions provided are

- `acedXrefAttach()`
- `acedXrefOverlay()`
- `acedXrefUnload()`
- `acedXrefDetach()`
- `acedXrefReload()`
- `acedXrefBind()`
- `acedXrefXBind()`
- `acedXrefCreateBlockname()`
- `acedXrefReload()`

For information on the AutoCAD XREF command, see the *AutoCAD User's Guide*.

The main programming consideration concerning xrefs is that, for every xref that is attached to a drawing, a separate database is created to represent the drawing containing the xref. A block table record in the main drawing contains the name of the external drawing and points to the entities of the model space of the externally referenced drawing. The xref database also contains other block table records and symbol table entries required to resolve all references from the main block table record (layers, linetypes, and so on).

You can create an editor reactor, as described in chapter 15, "Notification," to monitor xref events. The `AcEditorReactor` class provides the following reactor callback functions:

- `beginAttach()`
- `otherAttach()`
- `abortAttach()`
- `endAttach()`
- `redirected()`
- `comandeered()`

When using these functions, be careful to notice which database is being returned. Also, be aware that the xref drawing can itself contain xrefs to additional drawings. For more information on the `AcEditorReactor` class, see the *ObjectARX Reference*.

Xref entities in a drawing can be modified, but they cannot be saved to the original xref drawing (the original drawing is read-only).

External Reference Pre- and Post-Processing

External reference (xref) pre- and post-processing makes it possible to restore an attached xref's in-memory `AcDbDatabase` so that it can be saved back to a file. During xref resolve, many symbol table records are mangled, and some are erased. Historically, this was done to simplify the resolve process, and was acceptable because the databases were read-only. This processing makes it possible to temporarily reverse the resolve changes so that the xref database can be modified and written back to its file.

The functions that aid in pre- and post-processing are added to `AcDbDatabase`. They include a utility function to find the associated block table record from an xref database, as well as the ability to restore the resolved xref, and to reset it back to the proper resolved condition after restoration.

The customary usage for these functions would be to do the restore to the original symbols, make the modifications to the database, save the database, and then restore the forwarded symbols. These steps must be written into a single block of code, to prevent attempts to regenerate the host drawing, execute any xref commands, or provide user prompts while the xref database is in its restored condition.

The functions are

- `AcDbDatabase::xrefBlockId()`
- `AcDbDatabase::restoreOriginalXrefSymbols()`
- `AcDbDatabase::restoreForwardingXrefSymbols()`

AcDbDatabase::xrefBlockId() Function

The `xrefBlockId()` function will get the `AcDbObjectId` of the block table record, which refers to this database as an xref. When an xref is reloaded for any reason (for example, XREF Reload or XREF Path commands), the former database is kept in memory for Undo. This means that more than one database may point to the same xref block table record. However only one will be the currently active xref database for that record. The database pointer returned by the `AcDbBlockTableRecord::xrefDatabase()` function on the found record will be the active database for that xref.

AcDbDatabase::restoreOriginalXrefSymbols() Function

The `restoreOriginalXrefSymbols()` function restores a resolved xref database to its original form, as it would be if just loaded from its file. The xref is then in a condition where it can be modified and saved back to a file. After calling this function, the host drawing is no longer in a valid state for regen or for any xref command modifications or reloads. The database modifications, save back, and the `restoreForwardingXrefSymbols()` function must be called before anything that might allow a regen.

AcDbDatabase::restoreForwardingXrefSymbols() Function

The `restoreForwardingXrefSymbols()` function restores the xref back to a valid, attached state. Not only does it restore the original resolved symbols, but it also seeks out newly added symbols and resolves them as well. The `restoreForwardingXrefSymbols()` function cannot handle newly added, nested xref block table records unless they already exist and are resolved in the host drawing.

File Locking and Consistency Checks

The `AcDbXrefFileLock` base class is provided to handle the management of xref file locking. Its main purpose is to prepare the xref block in a drawing for in-place editing, though, it can be used for other purposes. It is assumed that these xref file methods operate on the current database drawing.

The `acdbXrefReload()` global function processes the list of xref block table record object IDs for xref reload. It is assumed that each xref block table record object ID references an xref drawing file that can be reloaded to the current drawing. It has the same functionality as the AutoCAD XREF subcommand for Reload.

Indexes and Filters

The index and filter classes and functions provide a scheme for applications to define custom indexes and custom filtering of block data. An application can define its custom implementations of `AcDbFilter`, `AcDbIndex`, and `AcDbFilteredBlockIterator`. It will register the `AcDbFilter` with a block reference through `AcIndexFilterManager::addFilter()`, and an `AcDbIndex` with the corresponding block table record through `AcIndexFilterManager::addIndex()`. After that, regens of xrefs and blocks will respect the query defined by the `AcDbFilter`, and use the `AcDbFilteredBlockIterator` to decide what object IDs will be processed during regen. The indexes will be kept up to date through either the application explicitly calling `AcIndexFilterManager::updateIndexes()`, or the application can rely on the AutoCAD save operation calling `AcIndexFilterManager::updateIndexes()` on the `AcDbDatabase` being saved.

The `AcDbIndex::rebuildFull()` or the `AcDbIndex::rebuildModified()` gets invoked during the `AcIndexFilterManager::updateIndexes()` call.

A current use of the indexing scheme in AutoCAD is fast demand loading of clipped xrefs. A spatial index (an `AcDbSpatialIndex` object) is stored in the xrefed drawing. An `AcDbSpatialFilter` object defines the clip volume of the block reference to the xref in the host drawing. When demand loading is turned on for the xref, the spatial filter volume is used to traverse the xref data through the spatial index, in order to page in from the DWG file only those entities whose graphics intersect the clip volume.

These classes and functions provide an interface for:

- Updating indexes
- Adding and removing indexes to block table records
- Adding and removing filters to block references
- Querying for indexes from block table records
- Querying for filters from block references
- Iterating through block table records and visiting only a subset of entities

The main classes and functions involved are

- `AcDbIndexFilterManager` namespace
- `AcDbIndex` class
- `AcDbFilter` class
- `AcDbFilteredBlockIterator` class
- `AcDbCompositeFilteredBlockIterator` class

AcDbIndexFilterManager Namespace

The `AcDbIndexFilterManager` namespace is a collection of functions that provides index and filter access and maintenance functionality.

AcDbIndex Class

The `AcDbIndex` class is the base class for all index objects. `AcDbSpatialIndex` and `AcDbLayerIndex` derive from this class.

Keeping the index up to date is achieved through the `AcDbIndexFilterManager::updateIndexes()` function calls being explicitly invoked (either by an application or AutoCAD).

The `AcDbFilteredBlockIterator` will serve as the means to visit all the `AcDbObjectIds` that are "hits" from the query defined by the `AcDbFilter` passed to its constructor. For example, in the spatial index case, the `AcDbSpatialFilter` object instance passed to the `newIterator()` method will define a query region. The `AcDbSpatialIndex` object, through its `newIterator()` method, will provide an `AcDbSpatialIndexIterator` that will return object IDs that correspond to entities that fit within the query volume.

AcDbFilter class

The `AcDbFilter` class is meant to define a "query." It provides the "key" to the `AcDbCompositeFilteredBlockIterator`, for which the corresponding index is obtained through the `indexClass()` method.

AcDbFilteredBlockIterator Class

The `AcDbFilteredBlockIterator` class provides a method to process a "query" on an index. It is used by the `AcDbCompositeFilteredBlockIterator`.

AcDbCompositeFilteredBlockIterator Class

The `AcDbCompositeFilteredBlockIterator` class provides the alternate to normal block iteration. By providing the filter list in the `init()` method, the `AcDbCompositeFilteredBlockIterator` object looks for corresponding `AcDbIndex` derived objects through the `AcDbFilter::indexClass()` method, and creates `AcDbFilteredBlockIterator` objects. If the matching up-to-date `indexClass()` objects are not available, it creates an `AcDbFilteredBlockIterator` through the `AcDbFilter::newIterator()` method. It then orders the composition of the `AcDbFilteredBlockIterator` objects based on the `AcDbFilteredBlockIterator::estimatedHits()` and `AcDbFilteredBlockIterator::buffersForComposition()` methods. The col-

lection of filters is a conjunction of conditions. This means an object ID is output from the iterator only if the `accepts()` method of each filter would accept the object ID.

Drawing Summary Information

The Drawing Property Dialog allows AutoCAD users to embed ancillary data (called summary information) in their DWG files, and assists in retrieving DWG files based on this data. This provides AutoCAD users with base-level file retrieval and management capabilities.

Through Windows Explorer, the properties of a drawing can be viewed outside of AutoCAD. Used in conjunction with the AutoCAD DesignCenter Advanced Find feature, summary information allows users to search for drawings containing predefined or custom data.

The `AcDbDatabaseSummaryInfo`, `AcDbSummaryInfoReactor`, and `AcDbSummaryInfoManager` classes provide an API to work with summary information and are discussed below. For more detail on these classes, see the *ObjectARX Reference*.

AcDbDatabaseSummaryInfo Class

The `AcDbDatabaseSummaryInfo` class encapsulates a set of character strings that can be used to add additional information to a DWG file. The maximum length of these strings is 511 characters. This information is stored and retrieved in the Summary Information object with specific methods for each information field. The predefined fields are

- Title
- Subject
- Author
- Keywords
- Comments
- Last saved by
- Revision number
- Hyperlink base

You can create your own custom fields in addition to the predefined fields. These custom fields are stored in a list, and you can manipulate custom fields either by their name (or key) or position (index) in the list. Custom fields are indexed starting at 1, and there is no limit to the number of fields you can create.

AcDbSummaryInfoReactor Class

This class provides a reactor to let you know if the summary information is changed.

AcDbSummaryInfoManager Class

The `AcDbSummaryInfoManager` class organizes the summary information reactors, with methods to add and remove reactors, and to send notification that the summary information has changed.

Global Summary Information Functions

ObjectARX contains several global functions for accessing summary information:

```
Acad::ErrorStatus
acdbGetSummaryInfo(
    AcDbDatabase* pDb,
    AcDbDatabaseSummaryInfo*& pInfo);
```

```
Acad::ErrorStatus
acdbPutSummaryInfo(
    const AcDbDatabaseSummaryInfo* pInfo);
```

```
AcDbSummaryInfoManager*
acdbGetSummaryInfoManager();
```

For more information on these functions, see the *ObjectARX Reference*.

Last Saved by Autodesk Software

The following `AcDbDatabase` method returns `Adesk::kTrue` if it determines that the database was last saved by Autodesk software (such as AutoCAD or AutoCAD LT®):

```
Adesk::Boolean
dwgFileWasSavedByAutodeskSoftware();
```

Database Objects

In This Chapter

5

- Opening and Closing Database Objects
- Deleting Objects
- Database Ownership of Objects
- Adding Object-Specific Data
- Erasing Objects
- Object Filing

This chapter describes topics that relate to all AutoCAD database objects, including entities, symbol table records, and dictionaries. Major concepts included are opening and closing objects, managing objects in memory, object ownership, and extending an object using xdata or the object's extension dictionary. Other common operations on objects, such as filing and erasing, are also discussed.

Opening and Closing Database Objects

Each `AcDbObject` object can be referred to in three different ways:

- By its handle
- By its object ID
- By a C++ instance pointer

When AutoCAD is not running, the drawing is stored in the file system. Objects contained in a DWG file are identified by their handles.

After the drawing is opened, the drawing information is accessible through the `AcDbDatabase` object. Each object in the database has an object ID, which persists throughout the current edit session, from creation until deletion of the `AcDbDatabase` in which the object resides. The open functions take an object ID as an argument and return a pointer to an `AcDbObject` object. This pointer is valid until the object is closed, as shown in the following figure.

You can open an object using the `acdbOpenObject()` function:

```
Acad::ErrorStatus
AcDbDatabase::acdbOpenObject(AcDbObject*& obj,
                             AcDbObjectId id,
                             AcDb::OpenMode mode,
                             Adesk::Boolean
                                 openErasedObject =
                                 Adesk::kFalse)
```

You can map a handle to an object ID using this function:

```
Acad::ErrorStatus
getAcDbObjectId(AcDbObjectId& retId,
                Adesk::Boolean createIfNotFound,
                const AcDbHandle& objHandle,
                Adesk::UInt32 xRefId=0);
```

You can also open an object and then request its handle:

```
AcDbObject* pObject;
AcDbHandle handle;

pObject->getAcDbHandle(handle);
```

NOTE Whenever a database object is opened, it should be closed at the earliest possible opportunity. You can use the `AcDbObject::close()` function to close a database object.

An `ads_name` is equivalent to an `AcDbObjectId`. The AcDb library provides two standalone functions that allow you to translate between an `AcDbObjectId` and an `ads_name`:

```
// Returns an ads_name for a given object ID.
//
acdbGetAdsName(ads_name& objName,
               AcDbObjectId objId);

// Returns an object ID for a given ads_name.
//
acdbGetObjectId(AcDbObjectId& objId,
                ads_name objName);
```

Generally, you obtain an object through a selection, and it is returned in `ads_name` form. You then need to exchange the `ads_name` for an `AcDbObjectId` and open it. The following function demonstrates this process:

```
AcDbEntity*
selectEntity(AcDbObjectId& eId, AcDb::OpenMode openMode)
{
    ads_name en;
    ads_point pt;
    acedEntSel("\nSelect an entity: ", en, pt);

    // Exchange the ads_name for an object ID.
    //
    acdbGetObjectId(eId, en);

    AcDbEntity * pEnt;
    acdbOpenObject(pEnt, eId, openMode);

    return pEnt;
}
```

You can open an object in one of three modes:

- `kForRead`. An object can be opened for read by up to 256 readers as long as the object is not already open for write or for notify.
- `kForWrite`. An object can be opened for write if it is not already open. Otherwise, the open fails.
- `kForNotify`. An object can be opened for notification when the object is closed, open for read, or open for write, but not when it is already open for notify. See chapter 15, "Notification." Applications will rarely need to open an object for notify and send it notification.

The following table shows the error codes returned when you attempt to open an object in different modes and the object is already open.

Opening objects in different modes

Object already opened for:	**kForRead**	**kForWrite**	**kForNotify**
openedForRead	eAtMaxReaders (if readCount = 256; otherwise succeeds)	eWasOpenForRead	(Succeeds)
openedForWrite	eWasOpenForWrite	eWasOpenForWrite	(Succeeds)
openedForNotify	eWasOpenForNotify	eWasOpenForNotify	eWasOpenForNotify
wasNotifying	(Succeeds)	eWasNotifying	eWasNotifying
Undo	eWasOpenForUndo	eWasOpenForUndo	(Succeeds)

If you are trying to open an object for write and you receive an error `eWasOpenForRead`, you can use `upgradeOpen()` to upgrade the open status to write if there is only one reader of the object. Then you would use `downgradeOpen()` to downgrade its status to read. Similarly, if your object is open for notify—for example, when you are receiving notification—and you want to open it for write, you can use `upgradeFromNotify()` to upgrade its open status to write. Then you would use `downgradeToNotify()` to downgrade its status to notify.

For more information about how to manage complex sequences of opening and closing objects, see "Transaction Manager" on page 451.

Deleting Objects

When you create an instance of an `AcDbObject` object with the intent of appending it to the database, use the `AcDbObject::new()` function. When an object is first created and has not yet been added to the database, you can delete it. However, once an object has been added to the database, you cannot delete it; AutoCAD manages the deletion of all database-resident objects.

Database Ownership of Objects

An object that is implicitly owned by the database rather than another database object is called a root object. The database contains ten root objects: the nine symbol tables and the named object dictionary. All filing operations begin by filing out the root objects of the database. See "Object Filing" on page 95.

With the exception of root objects, every object in the database must have an owner, and a given object can have only one owner. The database is a tree created by this hierarchy of owned objects. The following call adds an object to the database and assigns an ID to it, but the object does not yet have an owner:

```
db->addAcDbObject(...);
```

Usually, you will add the object to its owner using a member function that simultaneously adds it to the database, such as the `AcDbBlockTableRecord::appendAcDbEntity()` function, which performs both tasks at once.

AutoCAD ownership connections are as follows:

- The block table records own entities.
- Each symbol table owns a particular type of symbol table record.
- An `AcDbDictionary` object can own any `AcDbObject` object.
- Any `AcDbObject` object can have an extension dictionary; an object owns its extension dictionary.

In addition, applications can set up their own ownership connections.

Adding Object-Specific Data

You can use any of four mechanisms for adding instance-specific data in your application:

- Extended data (xdata)
- Xrecords (see chapter 7, "Container Objects")
- Extension dictionaries of any object
- Custom objects that can hold data (see chapter 12, "Deriving from AcDbObject")

Extended Data

Extended data (xdata) is created by applications written with ObjectARX or AutoLISP and can be added to any object. Xdata consists of a linked list of resbufs used by the application. (AutoCAD maintains the information but doesn't use it.) The data is associated with a DXF group code in the range of 1000 to 1071.

This mechanism is space-efficient and can be useful for adding lightweight data to an object. However, xdata is limited to 16K and to the existing set of DXF group codes and types.

For a more detailed description of xdata, see the *AutoCAD Customization Guide*.

Use the `AcDbObject::xData()` function to obtain the resbuf chain containing a copy of the xdata for an object:

```
virtual resbuf*
AcDbObject::xData(const char* regappName = NULL) const;
```

Use the `AcDbObject::setXData()` function to specify the xdata for an object:

```
virtual Acad::ErrorStatus
AcDbObject::setXData(const resbuf* xdata);
```

The following example uses the `xData()` function to obtain the xdata for a selected object and then prints the xdata to the screen. It then adds a string (`testrun`) to the xdata and calls the `setXdata()` function to modify the object's xdata. This example also illustrates the use of the `upgradeOpen()` and `downgradeOpen()` functions.

```
// This function calls the selectObject() function to allow
// the user to pick an object; then it accesses the xdata of
// the object and sends the list to the printList() function
// that lists the restype and resval values.
//
```

```
void
printXdata()
{
    // Select and open an object.
    //
    AcDbObject *pObj;
    if ((pObj = selectObject(AcDb::kForRead)) == NULL) {
        return;
    }

    // Get the application name for the xdata.
    //
    char appname[133];
    if (acedGetString(NULL,
        "\nEnter the desired Xdata application name: ",
        appname) != RTNORM)
    {
        return;
    }

    // Get the xdata for the application name.
    //
    struct resbuf *pRb;
    pRb = pObj->xData(appname);
    if (pRb != NULL) {
        // Print the existing xdata if any is present.
        // Notice that there is no -3 group, as there is in
        // LISP. This is ONLY the xdata, so
        // the -3 xdata-start marker isn't needed.
        //
        printList(pRb);
        acutRelRb(pRb);
    } else {
        acutPrintf("\nNo xdata for this appname");
    }
    pObj->close();
}

void
addXdata()
{
    AcDbObject* pObj = selectObject(AcDb::kForRead);
    if (!pObj) {
        acutPrintf("Error selecting object\n");
        return;
    }

    // Get the application name and string to be added to
    // xdata.
    //
```

```
    char appName[132], resString[200];
    appName[0] = resString[0] = '\0';
    acedGetString(NULL, "Enter application name: ",
        appName);
    acedGetString(NULL, "Enter string to be added: ",
        resString);

    struct  resbuf  *pRb, *pTemp;

    pRb = pObj->xData(appName);
    if (pRb != NULL) {
        // If xdata is present, then walk to the
        // end of the list.
        //
        for (pTemp = pRb; pTemp->rbnext != NULL;
                pTemp = pTemp->rbnext)
                { ; }
    } else {
        // If xdata is not present, register the application
        // and add appName to the first resbuf in the list.
        // Notice that there is no -3 group as there is in
        // AutoLISP. This is ONLY the xdata so
        // the -3 xdata-start marker isn't needed.
        //
        acdbRegApp(appName);
        pRb = acutNewRb(AcDb::kDxfRegAppName);
        pTemp = pRb;
        pTemp->resval.rstring
            = (char*) malloc(strlen(appName) + 1);
        strcpy(pTemp->resval.rstring, appName);
    }

    // Add user-specified string to the xdata.
    //
    pTemp->rbnext = acutNewRb(AcDb::kDxfXdAsciiString);
    pTemp = pTemp->rbnext;
    pTemp->resval.rstring
        = (char*) malloc(strlen(resString) + 1);
    strcpy(pTemp->resval.rstring, resString);

    // The following code shows the use of upgradeOpen()
    // to change the entity from read to write.
    //
    pObj->upgradeOpen();
    pObj->setXData(pRb);

    pObj->close();
    acutRelRb(pRb);
}
```

Extension Dictionary

Every object can have an extension dictionary, which can contain an arbitrary set of `AcDbObject` objects. Using this mechanism, several applications can attach data to the same object. The extension dictionary requires more overhead than xdata, but it also provides a more flexible mechanism with higher capacity for adding data.

For an example of using an extension dictionary to attach an arbitrary string to any `AcDbObject`, see the *edinvent* program in the *samples* directory.

ObjectARX Example

The following example shows instantiating an xrecord and adding it to an extension dictionary in the named object dictionary:

```
void
createXrecord()
{
    AcDbXrecord *pXrec = new AcDbXrecord;
    AcDbObject *pObj;
    AcDbObjectId dictObjId, xrecObjId;
    AcDbDictionary* pDict;

    pObj = selectObject(AcDb::kForWrite);
    if (pObj == NULL) {
        return;
    }

    // Try to create an extension dictionary for this
    // object.  If the extension dictionary already exists,
    // this will be a no-op.
    //
    pObj->createExtensionDictionary();

    // Get the object ID of the extension dictionary for the
    // selected object.
    //
    dictObjId = pObj->extensionDictionary();
    pObj->close();

    // Open the extension dictionary and add the new
    // xrecord to it.
    //
    acdbOpenObject(pDict, dictObjId, AcDb::kForWrite);
    pDict->setAt("ASDK_XREC1", pXrec, xrecObjId);
    pDict->close();
```

```
    // Create a resbuf list to add to the xrecord.
    //
    struct resbuf* head;
    ads_point testpt = {1.0, 2.0, 0.0};
    head = acutBuildList(AcDb::kDxfText,
        "This is a test Xrecord list",
        AcDb::kDxfXCoord, testpt,
        AcDb::kDxfReal, 3.14159,
        AcDb::kDxfAngle, 3.14159,
        AcDb::kDxfColor, 1,
        AcDb::kDxfInt16, 180,
        0);

    // Add the data list to the xrecord.  Notice that this
    // member function takes a reference to a resbuf NOT a
    // pointer to a resbuf, so you must dereference the
    // pointer before sending it.
    //
    pXrec->setFromRbChain(*head);
    pXrec->close();
    acutRelRb(head);
}

// The listXrecord() function gets the xrecord associated with the
// key "ASDK_XREC1" and lists out its contents by passing the resbuf
// list to the function printList().
//
void
listXrecord()
{
    AcDbObject *pObj;
    AcDbXrecord *pXrec;
    AcDbObjectId dictObjId;
    AcDbDictionary *pDict;
    pObj = selectObject(AcDb::kForRead);
    if (pObj == NULL) {
        return;
    }

    // Get the object ID of the object's extension dictionary.
    //
    dictObjId = pObj->extensionDictionary();
    pObj->close();

    // Open the extension dictionary and get the xrecord
    // associated with the key ASDK_XREC1.
    //
    acdbOpenObject(pDict, dictObjId, AcDb::kForRead);
    pDict->getAt("ASDK_XREC1", (AcDbObject*&)pXrec,
        AcDb::kForRead);
    pDict->close();
```

```
    // Get the xrecord's data list and then close the xrecord.
    //
    struct resbuf *pRbList;
    pXrec->rbChain(&pRbList);
    pXrec->close();

    printList(pRbList);
    acutRelRb(pRbList);
}
```

Global Function Example

The following example uses global ObjectARX functions to create an xrecord and add it to the dictionary associated with the key `ASDK_REC`.

```
int
createXrecord()
{
    struct resbuf *pXrec, *pEnt, *pDict, *pTemp, *pTemp2;
    ads_point dummy, testpt = {1.0, 2.0, 0.0};
    ads_name xrecname, ename, extDict = {0L, 0L};

    // Have the user select an entity.  Then get its data.
    //
    if (acedEntSel("\nselect entity: ", ename, dummy)
        != RTNORM)
    {
        acutPrintf("\nNothing selected");
        acedRetVoid();
        return RTNORM;
    }

    pEnt = acdbEntGet(ename);

    // Now check to see if the entity already has an
    // extension dictionary.
    //
    for (pTemp = pEnt; pTemp->rbnext != NULL;
        pTemp = pTemp->rbnext)
    {
        if (pTemp->restype == 102) {
            if (!strcmp("{ACAD_XDICTIONARY",
                pTemp->resval.rstring))
            {
              ads_name_set(pTemp->rbnext->resval.rlname, extDict);
                break;
            }
        }
    }
```

```
    // If no extension dictionary exists, add one.
    //
    if (extDict[0] == 0L) {
        pDict = acutBuildList(RTDXF0, "DICTIONARY", 100,
            "AcDbDictionary", 0);
        acdbEntMakeX(pDict, extDict);
        acutRelRb(pDict);

        pDict = acutBuildList(102, "{ACAD_XDICTIONARY", 360,
            extDict, 102, "}", 0);

        for (pTemp = pEnt; pTemp->rbnext->restype != 100;
            pTemp = pTemp->rbnext)
            { ; }
        for (pTemp2 = pDict; pTemp2->rbnext != NULL;
            pTemp2 = pTemp2->rbnext)
            { ; }

        pTemp2->rbnext = pTemp->rbnext;
        pTemp->rbnext = pDict;
        acdbEntMod(pEnt);
        acutRelRb(pEnt);
    }

    // At this point the entity has an extension dictionary.
    // Create a resbuf list of the xrecord's entity information
    // and data.
    //
    pXrec = acutBuildList(RTDXF0, "XRECORD",
        100, "AcDbXrecord",
        1, "This is a test Xrecord list", //AcDb::kDxfText
        10, testpt,                       //AcDb::kDxfXCoord
        40, 3.14159,                      //AcDb::kDxfReal
        50, 3.14159,                      //AcDb::kDxfAngle
        60, 1,                            //AcDb::kDxfColor
        70, 180,                          //AcDb::kDxfInt16
        0);

    // Create the xrecord with no owner set.  The xrecord's
    // new entity name will be placed into the xrecname
    // argument.
    //
    acdbEntMakeX (pXrec, xrecname);
    acutRelRb (pXrec);
    // Set the xrecord's owner to the extension dictionary
    //
    acdbDictAdd(extDict, "ASDK_XRECADS", xrecname);
    acedRetVoid();
    return RTNORM;
}

// Accesses the xrecord associated with the key ASDK_XRECADS in
// the extension dictionary of a user-selected entity.  Then
// list out the contents of this xrecord using the printList
// function.
//
```

```
int
listXrecord()
{
    struct resbuf *pXrec, *pEnt, *pTemp;
    ads_point dummy;
    ads_name ename, extDict = {0L, 0L};

    // Have the user select an entity; then get its data.
    //
    if (acedEntSel("\nselect entity: ", ename, dummy) != RTNORM) {
        acutPrintf("\nNothing selected");
        acedRetVoid();
        return RTNORM;
    }

    pEnt = acdbEntGet(ename);

    // Get the entity name of the extension dictionary.
    //
   for (pTemp = pEnt;pTemp->rbnext != NULL;pTemp = pTemp->rbnext) {
        if (pTemp->restype == 102) {
          if (!strcmp("{ACAD_XDICTIONARY", pTemp->resval.rstring)){
               ads_name_set(pTemp->rbnext->resval.rlname, extDict);
                 break;
             }
        }
    }

    if (extDict[0] == 0L) {
        acutPrintf("\nNo extension dictionary present.");
        return RTNORM;
    }

    pXrec = acdbDictSearch(extDict, "ASDK_XRECADS", 0);
    if(pXrec) {
        printList(pXrec);
        acutRelRb(pXrec);
    }

    acedRetVoid();
    return RTNORM;
}
```

Erasing Objects

Any object in the database can be erased with the following function:

```
Acad::ErrorStatus
AcDbObject::erase(Adesk::Boolean Erasing = Adesk::kTrue);
```

NOTE The `erase()` function has different results for database objects and entities, with consequences for unerasing them:

- When a database object is erased, information about that object is removed from the dictionary. If the object is unerased with `erase(kfalse)`, the information is not automatically reintroduced. You must use the `setAt()` function to add the information to the dictionary again.
- When an entity is erased, it is simply flagged as erased in the block table record. The entity can be unerased with `erase(kfalse)`.

By default, you cannot open an erased object with the `acdbOpenObject()` function. If you attempt to do so, the `eWasErased` error code will be returned.

```
extern Acad::ErrorStatus
acdbOpenObject(AcDbObject*&    obj,
                   AcDbObjectId    objId,
                   AcDb::OpenMode  openMode,
                   Adesk::Boolean  openErasedObject =
                                   Adesk::kFalse);
```

To open an erased object, use `kTrue` for the last parameter of the `acdbOpenObject()` function.

Container objects such as polylines and block table records usually provide the option of skipping erased elements when iterating over their contents. The default behavior is to skip erased elements.

Erased objects are not filed out to DWG or DXF files.

Object Filing

Object filing refers to the conversion process between an object's state and a single sequence of data, for purposes such as storing it on disk, copying it, or recording its state for an undo operation. Filing out is sometimes called serializing. Filing an object in is the process of turning a sequence of data back into an object, sometimes called deserializing.

Filing is used in several contexts in AutoCAD:

- Writing and reading DWG files (uses DWG format)
- Writing and reading DXF files (uses DXF format)
- Communicating among AutoCAD, AutoLISP, and ObjectARX (uses DXF format)
- Undo recording and restoring (uses DWG format)
- Copying operations such as INSERT, XREF, and COPY (uses DWG format)
- Paging (uses DWG format)

`AcDbObject` has two member functions for filing out: `dwgOut()` and `dxfOut()`, and two member functions for filing in: `dwgIn()` and `dxfIn()`. These member functions are primarily called by AutoCAD; object filing is almost never explicitly controlled by applications that use the database. However, if your application implements new database object classes, you'll need a more in-depth understanding of object filing. See chapter 12, "Deriving from AcDbObject."

The `dwg-` and `dxf-` prefixes indicate two fundamentally different data formats, the first typically used in writing to and from DWG files, and the second primarily for DXF files and AutoLISP `entget`, `entmake`, and `entmod` functions. The primary difference between the two formats is that for DWG filers (an object that writes data to a file), the data is not explicitly tagged. The DXF filers, in contrast, associate a data group code with every element of data in a published data format (see chapter 12, "Deriving from AcDbObject").

Entities

6

In This Chapter

- Entities Defined
- Entity Ownership
- AutoCAD Release 12 Entities
- Common Entity Properties
- Common Entity Functions
- Creating Instances of AutoCAD Entities
- Complex Entities
- Coordinate System Access
- Curve Functions
- Associating Hyperlinks with Entities

This chapter describes entities—database objects with a graphical representation. It lists the properties and operations all entities have in common. Examples show how to create blocks, inserts, and complex entities, and how to select and highlight subentities.

Entities Defined

An entity is a database object that has a graphical representation. Examples of entities include lines, circles, arcs, text, solids, regions, splines, and ellipses. The `AcDbEntity` class is derived from `AcDbObject`.

With a few exceptions, entities contain all necessary information about their geometry. A few entities contain other objects that hold their geometric information or attributes. Complex entities include the following:

- `AcDb2dPolyline`, which owns `AcDb2dPolylineVertex` objects
- `AcDb3dPolyline`, which owns `AcDb3dPolylineVertex` objects
- `AcDbPolygonMesh`, which owns `AcDbPolygonMeshVertex` objects
- `AcDbPolyFaceMesh`, which owns `AcDbPolyFaceMeshVertex` objects and `AcDbFaceRecord` objects
- `AcDbBlockReference`, which owns `AcDbAttribute` objects
- `AcDbMInsertBlock`, which owns `AcDbAttribute` objects

Examples of creating and iterating through complex entities are provided in "Complex Entities" on page 134.

Entity Ownership

Entities in the database normally belong to an `AcDbBlockTableRecord`. The block table in a newly created database has three predefined records, *MODEL_SPACE, *PAPER_SPACE, and *PAPER_SPACE0, which represent model space and the two pre-defined paper space layouts. Additional records are added whenever the user creates new blocks (block records), typically by issuing a BLOCK, HATCH, or DIMENSION command.

The ownership structure for database entities is as follows:

AutoCAD Release 12 Entities

The following entities were included in AutoCAD Release 12 and are declared in the *dbents.h* file. You cannot safely derive new classes from the following Release 12 entities:

- `AcDb2dPolyline`
- `AcDb3dPolyline`
- `AcDbPolygonMesh`
- `AcDbPolyFaceMesh`
- `AcDbSequenceEnd`
- `AcDbBlockBegin`
- `AcDbBlockEnd`
- `AcDbVertex`
- `AcDbFaceRecord`
- `AcDb2dVertex`
- `AcDb3dPolylineVertex`
- `AcDbPolygonMeshVertex`
- `AcDbPolyFaceMeshVertex`
- `AcDbMInsertBlock`

Common Entity Properties

All entities have a number of common properties and include member functions for setting and getting their values. These properties, which can also be set by user commands, are the following:

- Color
- Linetype
- Linetype scale
- Visibility
- Layer
- Line weight
- Plot style name

When you add an entity to a block table record, AutoCAD automatically invokes the `AcDbEntity::setDatabaseDefaults()` function, which sets the properties to their default values if you have not explicitly set them.

`AcDbViewport` acquires the settings of the current graphics window.

If a property has not been explicitly specified for an entity, the database's current value for that property is used. See chapter 4, "Database Operations," for a description of the member functions used for setting and getting the current property values associated with the database.

Entity Color

Entity color can be set and read as numeric index values ranging from 0 to 256, or by instances of `AcCmColor`, which is provided for future use by an expanded color model. Currently, AutoCAD uses color indexes only. The correct color index can be obtained from an instance of `AcCmColor` using the `AcCmColor::getColorIndex()` member function.

Color indexes 1 through 7 are used for standard colors, as shown in the following table:

Colors 1 to 7

Color Number	Color Name
1	Red
2	Yellow
3	Green

Colors 1 to 7 (*continued*)	
Color Number	**Color Name**
4	Cyan
5	Blue
6	Magenta
7	White or Black

Colors 8 through 255 are defined by the display device.

The following index values have special meanings:

0	Specifies BYBLOCK. Entities inherit the color of the current block reference that points to the block table record that the entity resides in, or black/white if the entity resides directly in the model space or paper space block table record.
256	Specifies BYLAYER. Entities assume the color of the entity's associated layer.
257	No color. Only present from the time an entity is first instantiated until its color is set to a value between 0 and 256, or the entity is added to the database and assumes the database's current color index.

If a color value is specified for an entity, the current database default color value is ignored. Use the following functions to set and query an entity color:

```
virtual Acad::ErrorStatus
AcDbEntity::setColorIndex(Adesk::UInt16 color);

Adesk::UInt16
AcDbEntity::colorIndex() const;
```

Entity Linetype

The linetype value points to a symbol table entry that specifies a series of dots and dashes used for drawing lines. When an entity is instantiated, its linetype is set to `NULL`. When the entity is added to the database, if a linetype has not been specified for the entity, the linetype is set to the database's current linetype value. This default value is stored in the CELTYPE system variable. Linetype can be specified by name, by a string, or by the object ID of an `AcDbLineTypeTableRecord` in the entity's target database.

Special linetype entries are as follows:

CONTINUOUS	Default linetype, which is automatically created in the linetype symbol table
BYLAYER	Linetype value of the entity's layer
BYBLOCK	Linetype value of the entity's surrounding block definition's current block reference

If a linetype value is specified for an entity, the current database default linetype value is ignored.

The following functions enable you to set the linetype for an entity, either by name or by object ID:

```
virtual Acad::ErrorStatus
AcDbEntity::setLinetype(const char* newVal);

virtual Acad::ErrorStatus
AcDbEntity::setLinetype(AcDbObjectId newVal);
```

This function returns the name of the current entity linetype:

```
char* AcDbEntity::linetype() const;
```

This function returns the object ID for the symbol table record specifying the linetype:

```
AcDbObjectId AcDbEntity::linetypeId() const;
```

Entity Linetype Scale

When an entity is first instantiated, its linetype scale is initialized to an invalid value. When the entity is added to the database, if a linetype scale has not been specified for the entity, it is set to the database's current linetype scale value. This database default value is stored in the CELTSCALE system variable.

Linetype Scale Specified Per Entity

If a linetype scale value is specified for an entity, the current database default linetype scale value is ignored.

The following functions allow you to set and inquire the linetype scale for an entity:

```
Acad::ErrorStatus
AcDbEntity::setLinetypeScale(double newVal);

double
AcDbEntity::linetypeScale() const;
```

Regenerating a Drawing

When an entity is regenerated, its effective linetype scale is a product of both the entity linetype scale and the global database linetype scale. For nonpaper space entities, the linetype scale is calculated as follows:

```
effltscale = ent->linetypeScale() *
             ent->database()->ltscale();
```

If PSLTSCALE is 1, the effective linetype scale is then applied to the appearance of the model space entity when viewed in paper space. If PSLTSCALE is 0, then all linetype scaling is performed with respect to model space views. See the *AutoCAD User's Guide* for further explanation of linetype scales.

Entity Visibility

If you specify that an entity is invisible, it will be invisible regardless of other settings in the database. Other factors can also cause an entity to be invisible. For example, an entity will not be displayed if its layer is turned off or frozen. The value of `AcDb::Visibility` can be either `kInvisible` or `kVisible`.

```
Acad::ErrorStatus
AcDbEntity::setVisibility(AcDb::Visibility newVal);

AcDb::Visibility
AcDbEntity::visibility() const;
```

Entity Layer

All entities have an associated layer. The database always contains at least one layer (layer 0). As with linetypes, you can specify a layer for an entity. If you don't specify a layer, the default database layer value is used for a new entity.

Each layer also has associated properties, which include frozen/thawed, on/off, locked/unlocked, color, linetype, and viewport (see chapter 7, "Container Objects"). When an entity's color or linetype is BYLAYER, the value of the layer property is used for the entity.

If a layer value is specified for an entity, the current database layer value is ignored.

The following functions enable you to set the layer for an entity, either by name or by object ID:

```
Acad::ErrorStatus
AcDbEntity::setLayer(const char* newVal);

Acad::ErrorStatus
AcDbEntity::setLayer(AcDbObjectId newVal);
```

This function returns the name of the current entity layer:

```
char* AcDbEntity::layer() const;
```

This function returns the object ID for the current layer (an object of type `AcDbLayerTableRecord`):

```
AcDbObjectId AcDbEntity::layerId() const;
```

Common Entity Functions

Entities also have a number of common functions, primarily intended for use by AutoCAD. This section provides general background on using some of these functions. For examples of implementing the functions for new classes, see chapter 13, "Deriving from AcDbEntity."

Common entity functions include the following:

- `intersectWith()` is used in trim, extend, fillet, chamfer, break, and object snap Intersection operations
- `transformBy()` is used to pass in a transform matrix that moves, scales, or rotates points in the object
- `getTransformedCopy()` creates a copy of the object and applies a transformation to it
- `getOsnapPoints()` returns the snap points and the kind of snap points
- `getGripPoints()` returns the grip points, which are a superset of the stretch points
- `getStretchPoints()` defaults to `getGripPoints()` and usually has the same implementation
- `moveStretchPointsAt()` is used by the AutoCAD STRETCH command to move specified points and defaults to `transformBy()`
- `moveGripPointsAt()` is used by AutoCAD grip editing to move specified points and defaults to `transformBy()`
- `worldDraw()` creates a view-independent geometric representation of an entity
- `viewportDraw()` creates a view-dependent geometric representation of an entity
- `draw()` queues up the entity and flushes the graphics queue so that the entity and anything else in the queue are drawn
- `list()` is used by the AutoCAD LIST command and produces `acutPrintf()` statements
- `getGeomExtents()` returns the corner points of a box that encloses the 3D extents of your entity
- `explode()` decomposes an entity into a set of simpler elements

- `getSubentPathsAtGsMarker()` returns the subentity paths that correspond to the given GS marker (see "GS Markers and Subentities" on page 109)
- `getGsMarkersAtSubentPath()` returns the GS marker that corresponds to the given subentity path
- `subentPtr()` returns a pointer corresponding to the given subentity path
- `highlight()` highlights the specified subentity (see "GS Markers and Subentities" on page 109)

Object Snap Points

Objects can have certain characteristic points defined for them, such as a center point, midpoint, or endpoint. When AutoCAD is acquiring points and is in Object Snap mode, it invokes the `getOsnapPoints()` function to acquire the relevant snap points for the specified Object Snap mode. The following table lists the possible Object Snap modes.

Object Snap modes

Mode	Description
`kOsModeEnd`	Endpoint
`kOsModeMid`	Midpoint
`kOsModeCen`	Center
`kOsModeNode`	Node
`kOsModeQuad`	Quadrant
`kOsModeIns`	Insertion
`kOsModePerp`	Perpendicular
`kOsModeTan`	Tangent
`kOsModeNear`	Nearest

The signature for `AcDbEntity::getOsnapPoints()` is

```
virtual Acad::ErrorStatus
AcDbEntity::getOsnapPoints(
    AcDb::OsnapMode     osnapMode,
    int                 gsSelectionMark,
    const AcGePoint3d&  pickPoint,
    const AcGePoint3d&  lastPoint,
    const AcGeMatrix3d& viewXform,
    AcGePoint3dArray&   snapPoints,
    AcDbIntArray&       geomIds) const;
```

The `geomIds` argument is not currently used. Intersection object snap does not use this function.

Transform Functions

The `AcDbEntity` class provides two transformation functions:

```
virtual Acad::ErrorStatus
AcDbEntity::transformBy(const AcGeMatrix3d& xform);

virtual Acad::ErrorStatus
AcDbEntity::getTransformedCopy(const AcGeMatrix3d& xform,
                               AcDbEntity*&  ent) const;
```

The `transformBy()` function modifies the entity using the specified matrix. In AutoCAD, it is called by the grip move, rotate, scale, and mirror modes. In some cases, however, applying the transformation requires that a new entity be created. In such cases, the `getTransformedCopy()` function is used so that the resulting entity can be an instance of a different class than the original entity.

When you explode a block reference that has been nonuniformly scaled, the `getTransformedCopy()` function is called on the entities in the block reference to create the new entities (see "Exploding Entities" on page 123).

Intersecting for Points

The `intersectWith()` function returns the points where an entity intersects another entity in the drawing. Input values for this function are the entity and the intersection type, which can be one of the following:

- `kOnBothOperands` (neither entity is extended)
- `kExtendThis`
- `kExtendArg`
- `kExtendBoth`

For example, suppose a drawing contains the three lines shown in the following illustration. Line1 is "this" and line3 is the argument entity. If the

intersection type is `kExtendThis`, point A is returned as the point where line1 ("this") would intersect line3 if line1 were extended. If the intersection type is `kExtendArgument` and line2 is the argument entity, no data is returned because, even if it were extended, line2 would not intersect line1. If the intersection type is `kExtendBoth` and line2 is the argument entity, point B is returned. If the intersection type is `kExtendNone` and line2 is the argument entity, no data is returned.

The `intersectWith()` function is an overloaded function with two forms. The second form takes an additional argument, which is a projection plane for determining the apparent intersection of two entities. These are the signatures for the `intersectWith()` function:

```
virtual Acad::ErrorStatus
AcDbEntity::intersectWith(
    const AcDbEntity* ent,
    AcDb::Intersect   intType,
    AcGePoint3dArray& points,
    int               thisGsMarker = 0,
    int               otherGsMarker = 0) const;

virtual Acad::ErrorStatus
AcDbEntity::intersectWith(
    const AcDbEntity* ent,
    AcDb::Intersect   intType,
    const AcGePlane&  projPlane,
    AcGePoint3dArray& points,
    int               thisGsMarker = 0,
    int               otherGsMarker = 0) const;
```

The returned points are always on the entity ("this"). Therefore, in cases of apparent intersection, the intersected points are projected back to the entity before they are returned.

Both versions of the `intersectWith()` function allow you to supply optional GS markers to optimize performance for this function. If the entity's `intersectWith()` function has implemented the use of GS markers, then

supplying GS markers can localize the intersection area and speed up the test. For example, in the following drawing, if the user selects one line of the polygon, passing in the GS marker for that line eliminates the need to test the other five lines of the polygon.

GS Markers and Subentities

To draw itself, every entity makes calls to graphics primitives such as polylines, circles, and arcs, contained in the AcGi library. Any class derived from `AcDbEntity` can associate a graphics system (GS) marker with the display vectors it uses to draw itself. Each entity subclass controls where it inserts its GS markers. When a user selects an entity, the GS marker is used to identify which part of the entity was picked.

Solids derived from `AcDb3dSolid` are composed of vertices, edges, and faces. Each of these elements can be identified by a GS marker. The creator of the entity class decides where GS markers should be inserted, depending on what is most natural for the entity. A box, for example, creates a GS marker for each line used to draw the box. A cylinder creates three GS markers—one for its top, bottom, and outside faces.

An entity is composed of subentities of the following type: vertex, edge, or face. Currently, the only entities that support subentities are bodies, regions, solids, and mlines. Use the `getSubentPathsAtGsMarker()` function to obtain the paths to the subentities that are associated with a particular GS marker. More than one subentity can be associated with a single marker. In the case of the box, for example, marker 4 identifies the lower front edge of the box. If you ask for the vertices associated with this marker, the two vertices that

form the endpoints of this line are returned. If you ask for the edges associated with this marker, one entity—the line—is returned. If you ask for the faces associated with this marker, data for the front face and the bottom face of the box are returned.

Subentity Path

A subentity path uniquely identifies a subentity within a particular entity in a drawing. This path, of class `AcDbFullSubentPath`, consists of an array of object IDs and a subentity ID object:

```
{AcDbObjectIdArray  mObjectIds;
 AcDbSubentId       mSubentId;
}
```

The array contains the object IDs that specify the path to the "main" entity. For example, a block reference (an entity that references a block table record) might contain two boxes, each of type `AcDb3dSolid`. The object ID array contains two entries: the ID of the block reference, followed by the ID of the main entity [InsertID, SolidID].

The second element of an `AcDbFullSubentPath` is an `AcDbSubentId` object, which has a subentity type (vertex, edge, or face) and the index of the subentity in the list. Use the `AcDbSubentId` functions `type()` and `index()` to access the member data.

Using the previous example, the second edge of the solid will have its `AcDbFullSubentPath` as

```
{(InsertID, solid1ID)
 (kEdgeSubentType, 2)};
```

If you have a solid only, `AcDbFullSubentPath` would be as follows for the first face of the solid.

```
{(solidID)
 (kFaceSubentType, 1)};
```

Simple Highlighting Example

The code example later in this section shows how to highlight a subentity. The following procedure lists the basic steps.

To highlight a subentity

1. Obtain the GS marker for the selected entity from the selection set.
2. Pass the GS marker to the entity class to be converted to a subentity path using the `getSubentPathsAtGsMarker()` function. Specify the type of subentity you're interested in (vertex, edge, face).
3. Once you have the path to the selected subentity, you're ready to call the `highlight()` function, passing in the correct subentity path.

Selecting an Entity

For selection, you'll use a combination of global functions. First, use the `acedSSGet()` function to obtain the selection set. Then, use the `acedSSNameX()` function to obtain a subentity GS marker for the selected entity.

```
int acedSSGet(
    const char *str,
    const void *pt1,
    const ads_point pt2,
    const struct resbuf *entmask,
    ads_name ss);

int acedSSNameX(
    struct resbuf** rbpp,
    const ads_name ss,
    const longvi);
```

Converting GS Markers to Subentity Paths

Use the `getSubentPathsAtGsMarker()` function to obtain the subentity for the GS marker returned by the `acedSSNameX()` function. The complete syntax for this function is

```
virtual Acad::ErrorStatus
AcDbEntity::getSubentPathsAtGsMarker(
    AcDb::SubentType type,
    int gsMark,
    const AcGePoint3d& pickPoint,
    const AcGeMatrix3d& viewXform,
    int& numPaths,
    AcDbFullSubentPath*& subentPaths
    int numInserts = 0,
    AcDbObjectId* entAndInsertStack = NULL) const;
```

The first argument to this function is the type of subentity you're interested in (vertex, edge, or face). In the example code in "Highlighting the Subentity," the first call to this function specifies `kEdgeSubentType` because you're

going to highlight the corresponding edge. The second call to the `getSubentPathsAtGsMarker()` function specifies `kFaceSubentType` because you're going to highlight each face associated with the selected subentity.

The `pickPoint` and `viewXform` arguments are used as additional input for some entities (such as mlines) when the GS marker alone does not provide enough information to return the subentity paths. In the example code in "Highlighting the Subentity," they are not used.

The `numInserts` and `entAndInsertStack` arguments are used for nested inserts. Both the `acedNEntSel()` and `acedNEntSelP()` functions return the name of the leaf-level entity, plus a stack of inserts.

Highlighting the Subentity

Once you've obtained the subentity path to the selected entity, the hardest part of this process is finished. Now, you need only call the `highlight()` function and pass in the subentity path. If you call the `highlight()` function without any arguments, the default is to highlight the whole entity.

The following sample code illustrates the steps described for selecting an entity, obtaining a subentity path, and highlighting different types of subentities associated with a GS marker. This code also illustrates another useful subentity function:

```
virtual AcDbEntity*
AcDbEntity::subentPtr(const AcDbFullSubentPath& id) const;
```

This function returns a pointer to a copy of the subentity described by the specified path, which can then be added to the database (as shown in the example).

NOTE It is expected that you will need to override the functions `getSubentPathsAtGsMarker()`, `getGsMarkersAtSubentPath()` and `subentPtr()` when you are creating new subclasses of `AcDbEntity`. The `highlight()` function, however, is implemented at the `AcDbEntity` level and is not generally expected to be overridden. However, if it is overridden, any new implementation of this function must call `AcDbEntity::highlight()` to perform the highlighting.

```
// This function calls getObjectAndGsMarker() to get the
// object ID of a solid and its gsmarker. It then calls
// highlightEdge(), highlightFaces(), and highlightAll() to
// highlight the selected edge, all faces surrounding that
// edge, and then the whole solid.
//
```

```
void
highlightTest()
{
    AcDbObjectId objId;
    int marker;

    if (getObjectAndGsMarker(objId, marker) != Acad::eOk)
        return;
    highlightEdge(objId, marker);
    highlightFaces(objId, marker);
    highlightAll(objId);
}
// This function uses acedSSGet() to let the user select a
// single entity.  It then passes this selection set to
// acedSSNameX() to get the gsmarker. Finally, the entity name
// in the selection set is used to obtain the object ID of
// the selected entity.
//
Acad::ErrorStatus
getObjectAndGsMarker(AcDbObjectId& objId, int& marker)
{
    ads_name sset;
    if (acedSSGet("_:S", NULL, NULL, NULL, sset) != RTNORM) {
        acutPrintf("\nacedSSGet has failed");
        return Acad::eInvalidAdsName;
    }

    // Get the entity from the selection set and its
    // subentity ID. This code assumes that the user
    // selected only one item, a solid.
    //
    struct resbuf *pRb;
    if (acedSSNameX(&pRb, sset, 0) != RTNORM) {
        acedSSFree(sset);
        return Acad::eAmbiguousOutput;
    }
    acedSSFree(sset);

    // Walk the list to the third item, which is the selected
    // entity's entity name.
    //
    struct resbuf *pTemp;
    int i;
    for (i=1, pTemp = pRb;i<3;i++, pTemp = pTemp->rbnext)
        { ; }
    ads_name ename;
    ads_name_set(pTemp->resval.rlname, ename);
```

```
    // Move on to the fourth list element, which is the gsmarker.
    //
    pTemp = pTemp->rbnext;
    marker = pTemp->resval.rint;
    acutRelRb(pRb);
    acdbGetObjectId(objId, ename);

    return Acad::eOk;
}

// This function accepts an object ID and a gsmarker.
// The object is opened, the gsmarker is used to get the
// AcDbFullSubentIdPath, which is then used to highlight
// and unhighlight the edge used to select the object.
// Next, the object's subentPtr() function is used to get
// a copy of the edge. This copy is then added to the
// database. Finally, the object is closed.
//
void
highlightEdge(const AcDbObjectId& objId, const int marker)
{
    char dummy[133]; // space for acedGetString pauses below
    AcDbEntity *pEnt;
    acdbOpenAcDbEntity(pEnt, objId, AcDb::kForRead);
    // Get the subentity ID for the edge that is picked
    //
    AcGePoint3d pickpnt;
    AcGeMatrix3d xform;
    int numIds;
    AcDbFullSubentPath *subentIds;

    pEnt->getSubentPathsAtGsMarker(AcDb::kEdgeSubentType,
        marker, pickpnt, xform, numIds, subentIds);
    // At this point the subentId's variable contains the
    // address of an array of AcDbFullSubentPath objects.
    // The array should be one element long, so the picked
    // edge's AcDbFullSubentPath is in subentIds[0].
    //
    // For objects with no edges (such as a sphere), the
    // code to highlight an edge is meaningless and must
    // be skipped.
    //
    if (numIds > 0) {
        // Highlight the edge.
        //
        pEnt->highlight(subentIds[0]);

        // Pause to let user see the effect.
        //
        acedGetString(0, "\npress <RETURN> to continue...",
            dummy);
```

```
        // Unhighlight the picked edge.
        //
        pEnt->unhighlight(subentIds[0]);

        // Get a copy of the edge, and add it to the database.
        //
        AcDbEntity *pEntCpy = pEnt->subentPtr(subentIds[0]);
        AcDbObjectId objId;
        addToModelSpace(objId, pEntCpy);
    }
    delete []subentIds;
    pEnt->close();
}

// This function accepts an object ID and a gsmarker.
// The object is opened, the gsmarker is used to get the
// AcDbFullSubentIdPath, which is then used to highlight
// and unhighlight faces that share the edge used to
// select the object.  The object is then closed.
//
void
highlightFaces(const AcDbObjectId& objId, const int marker)
{
    char dummy[133];
    AcDbEntity *pEnt;

    acdbOpenAcDbEntity(pEnt, objId, AcDb::kForRead);

    // Get the subentIds for the faces.
    //
    AcGePoint3d pickpnt;
    AcGeMatrix3d xform;
    int numIds;
    AcDbFullSubentPath *subentIds;

    pEnt->getSubentPathsAtGsMarker(AcDb::kFaceSubentType,
        marker, pickpnt, xform, numIds, subentIds);

    // Walk the subentIds list, highlighting each face subentity.
    //
    for (int i = 0;i < numIds; i++) {
        pEnt->highlight(subentIds[i]); // Highlight face.

        // Pause to let the user see the effect.
        //
        acedGetString(0, "\npress <RETURN> to continue...",
            dummy);
        pEnt->unhighlight(subentIds[i]);
    }
    delete []subentIds;
    pEnt->close();
}
```

```
// This function accepts an object ID. The object is opened,
// and its highlight() and unhighlight() functions are
// used with no parameters, to highlight and
// unhighlight the edge used to select the object. The
// object is then closed.
//
void
highlightAll(const AcDbObjectId& objId)
{
    char dummy[133];
    AcDbEntity *pEnt;

    acdbOpenAcDbEntity(pEnt, objId, AcDb::kForRead);

    // Highlight the whole solid.
    //
    pEnt->highlight();

    // Pause to let user see the effect.
    //
    acedGetString(0, "\npress <RETURN> to continue...",
        dummy);
    pEnt->unhighlight();
    pEnt->close();
}

Acad::ErrorStatus
addToModelSpace(AcDbObjectId &objId, AcDbEntity* pEntity)
{
    AcDbBlockTable *pBlockTable;
    AcDbBlockTableRecord *pSpaceRecord;

    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);
    pBlockTable->getAt(ACDB_MODEL_SPACE, pSpaceRecord,
        AcDb::kForWrite);

    pSpaceRecord->appendAcDbEntity(objId, pEntity);

    pBlockTable->close();
    pEntity->close();
    pSpaceRecord->close();

    return Acad::eOk;
```

Highlighting Nested Block References

The example that follows shows highlighting nested block references. As shown in the following figure, the example creates six entities: three polys (a custom entity) and three boxes. It also creates three block references (inserts). Insert 3 (`ins3`) is an insert of a block that contains `poly3` and `box3`. Insert 2 (`ins2`) is an insert of a block that contains `poly2`, `box2`, and `ins3`. Insert 1 (`ins1`) is an insert of a block that contains `poly1`, `box1`, and `ins2`.

After the inserts are created, the example highlights the different components.

```
void
createInsert()
{
    // Create a nested insert and try highlighting its
    // various subcomponents.
    //
    // There are six entities in total -- three polys and
    // three boxes (solids). We’ve named them: poly1, poly2,
    // poly3, and box1, box2, box3. We also have three
    // inserts: ins1, ins2, ins3.
    //
    // ins3 is an insert of a block that contains (poly3, box3)
    // ins2 is an insert of a block that contains (poly2, box2,
    // ins3).
    // ins1 is an insert of a block that contains (poly1, box1,
    // ins2).
    //
    // Let's create these entities first.
    //
    // Polys
    //
    AsdkPoly *poly1, *poly2, *poly3;
    AcGeVector3d norm(0, 0, 1);
    if ((poly1=new AsdkPoly)==NULL){
        acutPrintf("\nOut of Memory.");
        return;
    }
    if (poly1->set(AcGePoint2d(2, 8),AcGePoint2d(4, 8), 6, norm,
        "POLY1",0)!=Acad::eOk){
       acutPrintf("\nCannot create object with given parameters.");
        delete poly1;
        return;
    }
```

```
if ((poly2=new AsdkPoly)==NULL){
    acutPrintf("\nOut of Memory.");
    delete poly1;
    return;
}
if (poly2->set(AcGePoint2d(7, 8), AcGePoint2d(9, 8), 6, norm,
    "POLY2",0)!=Acad::eOk){
   acutPrintf("\nCannot create object with given parameters.");
    delete poly1;
    delete poly2;
    return;
}
if ((poly3=new AsdkPoly)==NULL){
    acutPrintf("\nOut of Memory.");
    delete poly1;
    delete poly2;
    return;
}
if (poly3->set(AcGePoint2d(12, 8),AcGePoint2d(14, 8), 6, norm,
    "POLY3",0)!=Acad::eOk){
   acutPrintf("\nCannot create object with given parameters.");
    delete poly1;
    delete poly2;
    delete poly3;
    return;
}
postToDb(poly1);
postToDb(poly2);
postToDb(poly3);

// Boxes
//
AcDb3dSolid *box1, *box2, *box3;
box1 = new AcDb3dSolid();
box2 = new AcDb3dSolid();
box3 = new AcDb3dSolid();

box1->createBox(2, 2, 2);
box2->createBox(2, 2, 2);
box3->createBox(2, 2, 2);

AcGeMatrix3d mat;

mat(0, 3) = 2; mat(1, 3) = 2;
box1->transformBy(mat);
mat(0, 3) = 7; mat(1, 3) = 2;
box2->transformBy(mat);
mat(0, 3) = 12; mat(1, 3) = 2;
box3->transformBy(mat);

postToDb(box1);
postToDb(box2);
postToDb(box3);
```

```
    // Inserts
    //
    // Arguments to BLOCK are:
    //      blockname,
    //      insert point,
    //      select objects,
    //      empty string for selection complete
    // Arguments to INSERT are:
    //      blockname,
    //      insertion point,
    //      xscale,
    //      yscale,
    //      rotation angle
    //
    acedCommand_command(RTSTR, "_globcheck", RTSHORT, 0, RTNONE);
    acedCommand(RTSTR, "BLOCK", RTSTR, "blk3", RTSTR, "0,0",
                RTSTR, "14,8", RTSTR, "11,1", RTSTR, "",
                RTNONE);
    acedCommand(RTSTR, "INSERT", RTSTR, "blk3", RTSTR,
                "0,0", RTSHORT, 1, RTSHORT, 1, RTSHORT,
                0, RTNONE);
    acedCommand(RTSTR, "BLOCK", RTSTR, "blk2", RTSTR, "0,0",
                RTSTR, "9,8", RTSTR, "6,1", RTSTR, "11,1",
                RTSTR, "", RTNONE);
    acedCommand(RTSTR, "INSERT", RTSTR, "blk2", RTSTR,
                "0,0", RTSHORT, 1, RTSHORT, 1, RTSHORT,
                0, RTNONE);
    acedCommand(RTSTR, "BLOCK", RTSTR, "blk1", RTSTR, "0,0",
                RTSTR, "4,8", RTSTR, "1,1", RTSTR, "6,1",
                RTSTR, "", RTNONE);
    acedCommand(RTSTR, "INSERT", RTSTR, "blk1", RTSTR,
                "0,0", RTSHORT, 1, RTSHORT, 1, RTSHORT,
                0, RTNONE);
    return;
}

void
hilitInsert()
{
    Adesk::Boolean interrupted = Adesk::kFalse;
    acutPrintf("\nSelect an insert");

    Acad::ErrorStatus es = Acad::eOk;
    AcDbEntity  *ent = NULL;
    AcDbEntity  *ent2 = NULL;
    AcDbBlockReference *blRef = NULL;
    AcDbObjectId objectId, blRefId;
    ads_name     ename, sset;
```

```
for (;;) {
    switch (acedSSGet(NULL, NULL, NULL, NULL, sset)) {
    case RTNORM:
    {
        struct resbuf *rb;
        if (acedSSNameX(&rb, sset, 0) != RTNORM) {
            acutPrintf("\n acedSSNameX failed");
            acedSSFree(sset);
            return;
        }

        int            sel_method;
        ads_name       subname;
        short          marker;
        AcGePoint3d    pickpnt;
        AcGeVector3d   pickvec;

        if (!extractEntityInfo(rb,
                         sel_method,
                         ename,
                         subname,
                         marker,
                         pickpnt,
                         pickvec)) {
            acutPrintf("\nextractEntityInfo failed");
            acedSSFree(sset);
            return;
        }

        acedSSFree(sset);

        assert(marker != 0);
        if (marker == 0) {
            acutPrintf("\nmarker == 0");
            return;
        }

        // Get the insert first.
        //
        AOK(acdbGetObjectId(blRefId, ename));
        AOK(acdbOpenAcDbEntity(ent, blRefId,
            AcDb::kForRead));
        assert(ent != NULL);
        blRef = AcDbBlockReference::cast(ent);
        if (blRef == NULL) {
            acutPrintf("\nNot an insert.");
            AOK(ent->close());
            continue;
        }
        struct resbuf *insStack;
        ads_point pickpoint;
        ads_matrix adsmat;
        pickpoint[0] = pickpnt[0];
        pickpoint[1] = pickpnt[1];
        pickpoint[2] = pickpnt[2];
```

```
// Now get details on the entity that was
// selected.
//
if (acedNEntSelP(NULL, ename, pickpoint, TRUE,
    adsmat, &insStack) != RTNORM)
{
    acutPrintf("\nFailure in acedNEntSelP");
    return;
}
assert(insStack != NULL);
AOK(acdbGetObjectId(objectId, ename));
AOK(acdbOpenAcDbEntity(ent2, objectId,
    AcDb::kForRead));
assert(ent2 != NULL);

// Make an array of AcDbObjectIds from the
// insertStack.  Don't use the "smart array"
// AcDbObjectIdArray class, because the
// getSubentPathsAtGsMarker() function expects argument
// eight to be of type AcDbObjectId*. Just
// make room for approximately 100 IDs in the array.
//
AcDbObjectId *idArray = new AcDbObjectId[100];
int count = 0;
struct resbuf *rbIter = insStack;
AcDbObjectId objId;
acdbGetObjectId(objId, ename);
idArray[count++] = objId;

while (rbIter != NULL) {
    ename[0] = rbIter->resval.rlname[0];
    ename[1] = rbIter->resval.rlname[1];
    acdbGetObjectId(objId, ename);
    idArray[count++] = objId;
    rbIter = rbIter->rbnext;
}

count--;
acutRelRb(insStack);

// First, we'll highlight an edge.
//
int                  numPaths;
AcDbFullSubentPath  *subentPaths;
AcGeMatrix3d         xform;
es = blRef->getSubentPathsAtGsMarker(
    AcDb::kEdgeSubentType,
    marker,
    pickpnt,
    xform,
    numPaths,
    subentPaths,
    count,
    idArray);
assert(numPaths == 1);
```

```
// Highlight and unhighlight the selected edge.
//
acutPrintf("\nHighlighting the first edge.");
es = blRef->highlight(subentPaths[0]);
pressEnterToContinue();
es = blRef->unhighlight(subentPaths[0]);

// If this is a solid, it will have faces.
// In this case, let's highlight them.
//
if(ent2->isKindOf(AcDb3dSolid::desc())) {
    es = blRef->getSubentPathsAtGsMarker(
        AcDb::kFaceSubentType,
        marker,
        pickpnt,
        xform,
        numPaths,
        subentPaths,
        count,
        idArray);
    assert(numPaths == 2);

    // Highlight and unhighlight the selected
    // faces.
    //
    acutPrintf("\nHighlighting the first"
        " face.");
    es = blRef->highlight(subentPaths[0]);
    pressEnterToContinue();
    es = blRef->unhighlight(subentPaths[0]);
    acutPrintf("\nHighlighting the next face.");
    es = blRef->highlight(subentPaths[1]);
    pressEnterToContinue();
    es = blRef->unhighlight(subentPaths[1]);
}
delete []subentPaths;

// Now, let's highlight the whole entity.
//
acutPrintf("\nHighlighting the entire entity");

AcDbFullSubentPath subPath;
for (int i = count; i >= 0; i--) {
    subPath.objectIds().append(idArray[i]);
}
es = blRef->highlight(subPath);
pressEnterToContinue();
es = blRef->unhighlight(subPath);

// Finally, let's highlight each enclosing
// insert.
//
for (i = count -1; i >= 0; i --) {
    subPath.objectIds().removeAt(
        subPath.objectIds().length() - 1);
    acutPrintf("\nHighlighting insert layer %d",
```

```
                    i + 1);
                blRef->highlight(subPath);
                pressEnterToContinue();
                es = blRef->unhighlight(subPath);
            }
        }       // case RTNORM
            break;
        case RTNONE:
        case RTCAN:
            return;
        default:
            continue;
        } // switch
        break;
    } //for (;;)
    AOK(ent->close());
    AOK(ent2->close());
    return;
}
```

Exploding Entities

Some entities can be exploded, or decomposed, into a set of simpler elements. The specific behavior depends on the class. For example, boxes can be exploded into regions, then lines. Polylines can be exploded into line segments. An mtext entity can be exploded into a separate text entity for each line of the original object. An mline entity can be exploded into individual lines. When you explode a block reference, AutoCAD copies all entities in the block reference and then splits them into their components.

The `explode()` function creates an array of objects derived from `AcDbEntity`.

The following table shows what happens when you explode each entity, when it is by itself and when it is in a block insert that is nonuniformly scaled.

Exploding entities

Entity	By Itself	Nonuniform Scaling (when in a block)
AcDb3dSolid	Regions, bodies	NA; can't be exploded
AcDbBody	Regions, bodies	NA
Ac2dDbPolyline	Lines, arcs	Self/NA
Ac3dPolyline	Lines	Self
AcDbArc	Self	Ellipse

Exploding entities (*continued*)

Entity	By Itself	Nonuniform Scaling (when in a block)
AcDbCircle	Self	Ellipse
AcDbDimension	Solids, lines, text strings, points	NA
AcDbEllipse	Self	Self
AcDbLeader	Self	NA
AcDbLine	Self	Self
AcDbRay	Self	Self
AcDbSpline	Self	Self
AcDbXline	Self	Self
AcDbFace	Self	Self
AcDbMline	Lines	Self
AcDbMText	One text entity for each line	Self
AcDbPoint	Self	Self
AcDbPolyFaceMesh	AcDbFace	Self
AcDbPolygonMesh	Self	Self
AcDbRegion	Curves (splines, lines, arcs, circles)	NA
AcDbShape	Self	Self
AcDbSolid	Self	Self
AcDbText	Self	Self
AcDbTrace	Self	Self

The `explode()` function is a read-only function that does not modify the original entity. It returns a set of entities for the application to handle as desired. One potential use of this function is to explode a complex entity to

produce simpler entities and then operate on those entities. For example, if you were implementing an `intersectForPoints()` function for a polyline, it might be easier to deal with the individual pieces of the polyline rather than the complete entity.

The following statements are true for the EXPLODE command (but not for the `explode()` function):

- Visual appearance is constant.
- The entity being exploded is erased from the database.
- One or more new entities are created and appended to the database.

Creating Instances of AutoCAD Entities

This section demonstrates how to create simple and complex entities and add them to the database. It also illustrates creating a simple entity, a simple block, a block with attributes, and a block insert (a block reference).

Creating a Simple Entity

The following example demonstrates creating a line and appending it to the model space block table record, as described in chapter 2, "Database Primer."

```
AcDbObjectId
createLine()
{
    AcGePoint3d startPt(4.0, 2.0, 0.0);
    AcGePoint3d endPt(10.0, 7.0, 0.0);
    AcDbLine *pLine = new AcDbLine(startPt, endPt);

    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);
    pBlockTable->close();

    AcDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);

    pBlockTableRecord->close();
    pLine->close();

    return lineId;
}
```

Creating a Simple Block Table Record

The following example demonstrates creating a new block table record and appending it to the block table. Then it creates a line and appends it to the new block table record.

```
void
makeABlock()
{
     // Create and name a new block table record.
     //
     AcDbBlockTableRecord *pBlockTableRec
         = new AcDbBlockTableRecord();
     pBlockTableRec->setName("ASDK-NO-ATTR");

     // Get the block table.
     //
     AcDbBlockTable *pBlockTable = NULL;
     acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForWrite);

     // Add the new block table record to the block table.
     //
     AcDbObjectId blockTableRecordId;
     pBlockTable->add(blockTableRecordId, pBlockTableRec);
     pBlockTable->close();

     // Create and add a line entity to the component's
     // block record.
     //
     AcDbLine *pLine = new AcDbLine();
     AcDbObjectId lineId;
     pLine->setStartPoint(AcGePoint3d(3, 3, 0));
     pLine->setEndPoint(AcGePoint3d(6, 6, 0));
     pLine->setColorIndex(3);

     pBlockTableRec->appendAcDbEntity(lineId, pLine);
     pLine->close();
     pBlockTableRec->close();
}
```

Creating a Block Table Record with Attribute Definitions

An AutoCAD block is a collection of entities that is stored in a block table record. Each block has an `AcDbBlockBegin` object, followed by one or more `AcDbEntity` objects, and ends with an `AcDbBlockEnd` object (see the illustration on page 100).

A block can contain attribute definitions, which are templates for creating attributes. An attribute is informational text associated with a block. Depend-

ing on a user-supplied setting, attribute values may or may not be copied when a block is inserted into a drawing. Often, the application prompts the user for the attribute value at runtime.

To create a block table record

1. Create a new block table record.
2. Add the block table record to the block table.
3. Create entities and add them to the block table record.
4. Create attribute definitions, set their values, and add them to the block table record.

When you close the block table record, the block begin and block end objects are added to the block automatically.

The following example creates a new block table record named `ASDK-BLOCK-WITH-ATTR` and adds it to the block table. Next it creates a circle entity and adds it to the new block table record. It creates two attribute definition entities (the second is a clone of the first) and appends them to the same block table record.

```
void
defineBlockWithAttributes(
    AcDbObjectId& blockId, // This is a returned value.
    const AcGePoint3d& basePoint,
    double textHeight,
    double textAngle)
{
    int retCode = 0;
    AcDbBlockTable *pBlockTable = NULL;
    AcDbBlockTableRecord* pBlockRecord = new AcDbBlockTableRecord;
    AcDbObjectId entityId;
    // Step 1: Set the block name and base point of the
    // block definition.
    //
    pBlockRecord->setName("ASDK-BLOCK-WITH-ATTR");
    pBlockRecord->setOrigin(basePoint);

    // Open the block table for write.
    //
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForWrite);

    // Step 2: Add the block table record to block table.
    //
    pBlockTable->add(blockId, pBlockRecord);
```

```
// Step 3: Create a circle entity.
//
AcDbCircle *pCircle = new AcDbCircle;
pCircle->setCenter(basePoint);
pCircle->setRadius(textHeight * 4.0);
pCircle->setColorIndex(3);

// Append the circle entity to the block record.
//
pBlockRecord->appendAcDbEntity(entityId, pCircle);
pCircle->close();

// Step 4: Create an attribute definition entity.
//
AcDbAttributeDefinition *pAttdef
    = new AcDbAttributeDefinition;

// Set the attribute definition values.
//
pAttdef->setPosition(basePoint);
pAttdef->setHeight(textHeight);
pAttdef->setRotation(textAngle);
pAttdef->setHorizontalMode(AcDb::kTextLeft);
pAttdef->setVerticalMode(AcDb::kTextBase);
pAttdef->setPrompt("Prompt");
pAttdef->setTextString("DEFAULT");
pAttdef->setTag("Tag");
pAttdef->setInvisible(Adesk::kFalse);
pAttdef->setVerifiable(Adesk::kFalse);
pAttdef->setPreset(Adesk::kFalse);
pAttdef->setConstant(Adesk::kFalse);
pAttdef->setFieldLength(25);

// Append the attribute definition to the block.
//
pBlockRecord->appendAcDbEntity(entityId, pAttdef);

// The second attribute definition is a little easier
// because we are cloning the first one.
//
AcDbAttributeDefinition *pAttdef2
    = AcDbAttributeDefinition::cast(pAttdef->clone());

// Set the values that are specific to the
// second attribute definition.
//
AcGePoint3d tempPt(basePoint);
tempPt.y -= pAttdef2->height();
pAttdef2->setPosition(tempPt);
pAttdef2->setColorIndex(1); // Red
pAttdef2->setConstant(Adesk::kTrue);
```

```
    // Append the second attribute definition to the block.
    //
    pBlockRecord->appendAcDbEntity(entityId, pAttdef2);
    pAttdef->close();
    pAttdef2->close();
    pBlockRecord->close();
    pBlockTable->close();
    return;
}
```

Creating a Block Reference with Attributes

A block reference is an entity that references a block table record. It contains an insertion point, ECS information, *X,Y,Z* scale factors, rotation, and a normal vector (parameters for viewing the block in its new location). When you insert a block into a drawing, AutoCAD conserves memory by creating a block reference rather than copying the block itself into the drawing.

If you insert a block with attribute definitions, the attribute values can be filled in by the user at runtime or by the application when the block is inserted.

To insert a block with attributes into a drawing

1. Create a block reference entity (`AcDbBlockReference`).
2. Call the `setBlockTableRecord()` function to specify the object ID of the referenced block table record. (The object ID can also be specified directly in the constructor of the block reference.)
3. Append the block reference to a block table record (model space, paper space, or some other block).
4. Use a block table record iterator on the referenced block table record, searching for attribute definitions. For each one found, create a new `AcDbAttribute` entity, fill it in with the attribute definition's data, and then append it to the block reference using the `appendAttribute()` function.

The following example creates a block reference, fills in the attributes, and appends the reference to the database. It uses global functions to obtain user input. The `createBlockWithAttributes()` function shown in the previous section is used to create the block reference. This example uses a block table record iterator to step through the attribute definitions and create a corresponding attribute for each attribute definition. The attribute values are set from the original attribute definition using the `setPropertiesFrom()` function.

```
void
addBlockWithAttributes()
{
    // Get an insertion point for the block reference,
    // definition, and attribute definition.
    //
    AcGePoint3d basePoint;
    if (acedGetPoint(NULL, "\nEnter insertion point: ",
        asDblArray(basePoint)) != RTNORM)
        return;

    // Get the rotation angle for the attribute definition.
    //
    double textAngle;
    if (acedGetAngle(asDblArray(basePoint),
        "\nEnter rotation angle: ", &textAngle) != RTNORM)
        return;

    // Define the height used for the attribute definition text.
    //
    double textHeight;
    if (acedGetDist(asDblArray(basePoint),
        "\nEnter text height: ", &textHeight) != RTNORM)
        return;

    // Build the block definition to be inserted.
    //
    AcDbObjectId blockId;
    defineBlockWithAttributes(blockId, basePoint,
        textHeight, textAngle);

    // Step 1: Allocate a block reference object.
    //
    AcDbBlockReference *pBlkRef = new AcDbBlockReference;

    // Step 2: Set up the block reference to the newly
    // created block definition.
    //
    pBlkRef->setBlockTableRecord(blockId);
```

```
// Give it the current UCS normal.
//
struct resbuf to, from;
from.restype = RTSHORT;
from.resval.rint = 1; // UCS
to.restype = RTSHORT;
to.resval.rint = 0; // WCS
AcGeVector3d normal(0.0, 0.0, 1.0);
acedTrans(&(normal.x), &from, &to, Adesk::kTrue,
    &(normal.x));

// Set the insertion point for the block reference.
//
pBlkRef->setPosition(basePoint);

// Indicate the LCS 0.0 angle, not necessarily the UCS 0.0 angle.
//
pBlkRef->setRotation(0.0);
pBlkRef->setNormal(normal);

// Step 3: Open the current database's model space
// block Table Record.
//
AcDbBlockTable *pBlockTable;
acdbHostApplicationServices()->workingDatabase()
    ->getSymbolTable(pBlockTable, AcDb::kForRead);
AcDbBlockTableRecord *pBlockTableRecord;
pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
    AcDb::kForWrite);
pBlockTable->close();

// Append the block reference to the model space
// block Table Record.
//
AcDbObjectId newEntId;
pBlockTableRecord->appendAcDbEntity(newEntId, pBlkRef);
pBlockTableRecord->close();

// Step 4: Open the block definition for read.
//
AcDbBlockTableRecord *pBlockDef;
acdbOpenObject(pBlockDef, blockId, AcDb::kForRead);

// Set up a block table record iterator to iterate
// over the attribute definitions.
//
```

```
    AcDbBlockTableRecordIterator *pIterator;
    pBlockDef->newIterator(pIterator);
    AcDbEntity *pEnt;
    AcDbAttributeDefinition *pAttdef;
    for (pIterator->start(); !pIterator->done();
        pIterator->step())
    {
        // Get the next entity.
        //
        pIterator->getEntity(pEnt, AcDb::kForRead);

        // Make sure the entity is an attribute definition
        // and not a constant.
        //
        pAttdef = AcDbAttributeDefinition::cast(pEnt);
        if (pAttdef != NULL && !pAttdef->isConstant()) {
            // We have a non-constant attribute definition,
            // so build an attribute entity.
            //
            AcDbAttribute *pAtt = new AcDbAttribute();
            pAtt->setPropertiesFrom(pAttdef);
            pAtt->setInvisible(pAttdef->isInvisible());

            // Translate the attribute by block reference.
            // To be really correct, the entire block
            // reference transform should be applied here.
            //
            basePoint = pAttdef->position();
            basePoint += pBlkRef->position().asVector();
            pAtt->setPosition(basePoint);
            pAtt->setHeight(pAttdef->height());
            pAtt->setRotation(pAttdef->rotation());
            pAtt->setTag("Tag");
            pAtt->setFieldLength(25);
            char *pStr = pAttdef->tag();
            pAtt->setTag(pStr);
            free(pStr);
            pAtt->setFieldLength(pAttdef->fieldLength());

            // The database column value should be displayed.
            // INSERT prompts for this.
            //
            pAtt->setTextString("Assigned Attribute Value");
            AcDbObjectId attId;
            pBlkRef->appendAttribute(attId, pAtt);
            pAtt->close();
        }
        pEnt->close(); // use pEnt... pAttdef might be NULL
    }
    delete pIterator;
    pBlockDef->close();
    pBlkRef->close();
}
```

Iterating through a Block Table Record

The following example demonstrates how to iterate through the elements in a block table record and print out the elements.

The `printAll()` function opens the block table for reading, and then it opens the block name supplied by the user. A new iterator steps through the block table records. If the record contains an entity, the iterator prints a message about the entity.

```
void
printAll()
{
    int rc;
    char blkName[50];
    rc = acedGetString(Adesk::kTrue,
        "Enter Block Name <CR for current space>: ",
        blkName);
    if (rc != RTNORM)
        return;
    if (blkName[0] == '\0') {
        if (acdbHostApplicationServices()->workingDatabase()
            ->tilemode() == Adesk::kFalse) {
            struct resbuf rb;
            acedGetVar("cvport", &rb);
            if (rb.resval.rint == 1) {
                strcpy(blkName, ACDB_PAPER_SPACE);
            } else {
                strcpy(blkName, ACDB_MODEL_SPACE);
            }
        } else {
            strcpy(blkName, ACDB_MODEL_SPACE);
        }
    }

    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(blkName, pBlockTableRecord,
        AcDb::kForRead);
    pBlockTable->close();

    AcDbBlockTableRecordIterator *pBlockIterator;
    pBlockTableRecord->newIterator(pBlockIterator);
```

```
    for (; !pBlockIterator->done();
        pBlockIterator->step())
    {
        AcDbEntity *pEntity;
        pBlockIterator->getEntity(pEntity, AcDb::kForRead);

        AcDbHandle objHandle;
        pEntity->getAcDbHandle(objHandle);
        char handleStr[20];
        objHandle.getIntoAsciiBuffer(handleStr);
        const char *pCname = pEntity->isA()->name();

        acutPrintf("Object Id %lx, handle %s, class %s.\n",
            pEntity->objectId(), handleStr, pCname);
        pEntity->close();
    }
    delete pBlockIterator;
    pBlockTableRecord->close();
    acutPrintf("\n");
}
```

Complex Entities

This section provides examples showing how to create and iterate through complex entities.

Creating a Complex Entity

This example shows how to create an `AcDb2dPolyline` object and set some of its properties—layer, color index, the closed parameter. It then creates four vertex objects (`AcDb2dPolylineVertex`), sets their location, and appends them to the polyline object. Finally, it closes all the open objects—vertices, polyline, block table record, and block table. When the polyline object is closed, AutoCAD adds the `AcDbSequenceEnd` object to it automatically.

```
void
createPolyline()
{
    // Set four vertex locations for the pline.
    //
    AcGePoint3dArray ptArr;
    ptArr.setLogicalLength(4);
    for (int i = 0; i < 4; i++) {
        ptArr[i].set((double)(i/2), (double)(i%2), 0.0);
    }

    // Dynamically allocate an AcDb2dPolyline object,
    // given four vertex elements whose locations are supplied
    // in ptArr.  The polyline has no elevation, and is
    // explicitly set as closed.  The polyline is simple;
```

```
    // that is, not curve fit or a spline.  By default, the
    // widths are all 0.0 and there are no bulge factors.
    //
    AcDb2dPolyline *pNewPline = new AcDb2dPolyline(
        AcDb::k2dSimplePoly, ptArr, 0.0, Adesk::kTrue);
    pNewPline->setColorIndex(3);

    // Get a pointer to a Block Table object.
    //
    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    // Get a pointer to the MODEL_SPACE BlockTableRecord.
    //
    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);
    pBlockTable->close();

    // Append the pline object to the database and
    // obtain its object ID.
    //
    AcDbObjectId plineObjId;
    pBlockTableRecord->appendAcDbEntity(plineObjId,
        pNewPline);
    pBlockTableRecord->close();

    // Make the pline object reside on layer "0".
    //
    pNewPline->setLayer("0");
    pNewPline->close();
}
```

Iterating through Vertices in a Polyline

The following example shows iterating through the vertices in a polyline using a vertex iterator. It then prints the coordinates for each vertex.

```
// Accepts the object ID of an AcDb2dPolyline, opens it, and gets
// a vertex iterator. It then iterates through the vertices,
// printing out the vertex location.
//
void
iterate(AcDbObjectId plineId)
{
    AcDb2dPolyline *pPline;
    acdbOpenObject(pPline, plineId, AcDb::kForRead);

    AcDbObjectIterator *pVertIter= pPline->vertexIterator();
    pPline->close();  // Finished with the pline header.
```

```
    AcDb2dVertex *pVertex;
    AcGePoint3d location;
    AcDbObjectId vertexObjId;

    for (int vertexNumber = 0; !pVertIter->done();
        vertexNumber++, pVertIter->step())
    {
        vertexObjId = pVertIter->objectId();
        acdbOpenObject(pVertex, vertexObjId,
            AcDb::kForRead);

        location = pVertex->position();
        pVertex->close();

        acutPrintf("\nVertex #%d's location is"
            " : %0.3f, %0.3f, %0.3f", vertexNumber,
            location[X], location[Y], location[Z]);
    }
    delete pVertIter;
}
```

Coordinate System Access

Entity functions retrieve and set coordinate values using World Coordinate System values. The only exception to this rule is the `AcDb2dPolylineVertex` class, described later in this section, which uses Entity Coordinate System (ECS) values. For example, if you call the `getCenter()` function on a circle, AutoCAD returns the *X,Y* center of the circle in world coordinates.

Entity Coordinate System

If you define your own entity, it may be useful to store its geometric constructs (points, angles, and vectors) in terms of its own relative coordinate system. For example, arcs establish a coordinate system in which the *Z* axis is perpendicular to the plane of the arc. An arc's center point is returned in world coordinates, but the start and end angles can only be interpreted with respect to its ECS. In such cases, implement the `getEcs()` function to return a matrix that is used to transform the entity from its Entity Coordinate System to the World Coordinate System. If an entity is not defined in terms of its own Entity Coordinate System, then the `getEcs()` function returns the identity matrix. (In other words, any time an entity's `getEcs()` function returns the identity matrix, you can assume the entity is defined in terms of world coordinates.)

In AutoCAD, planar entities have an ECS; 3D entities do not. AutoCAD entities that can return a nonidentity matrix for their `getEcs()` function are:

- Dimensions
- Text
- Circles
- Arcs
- 2D polylines
- Block inserts
- Points
- Traces
- Solids
- Shapes
- Attribute definitions
- Attributes

AcDb2dPolylineVertex

An `AcDb2dPolyline` has as an elevation and a series of *X,Y* points of class `AcDb2dPolylineVertex`. The `position()` and `setPosition()` functions of `AcDb2dPolylineVertex` specify 3D locations in the ECS. The *Z* coordinate passed in to the `setPosition()` function is stored in the entity and is returned by the `position()` function, but is otherwise ignored. It does not affect the polyline's elevation.

The `AcDb2dPolyline` class provides the `vertexPosition()` function, which returns a World Coordinate System value for the vertex passed in. The only way to change the elevation of a polyline is using the `AcDb2dPolyline::setElevation()` function.

Curve Functions

The abstract base class `AcDbCurve` provides a number of functions for operating on curves, including functions for projecting, extending, and offsetting curves, as well as a set of functions for querying curve parameters. Curves can be defined either in parameter space or in Cartesian coordinate space. A 3D curve is a function of one parameter (*f(t)*), while a 3D surface is a function of two parameters (*f(u,v)*). Conversion functions allow you to convert data from its parameter representation to points in the Cartesian coordinate system. Splines, for example, are best represented in parameter space. To split a spline into three equal parts, you first find the parameters that correspond to the points of the spline and then operate on the spline in parameter space.

Curves can be used as trim boundaries, extension boundaries, and as construction objects for creating complex 3D entities.

You can project a curve onto a plane in a given direction, as shown in the following example.

```
// Accepts an ellipse object ID, opens the ellipse, and uses
// its getOrthoProjectedCurve member function to create a
// new ellipse that is the result of a projection onto the
// plane with normal <1,1,1>.  The resulting ellipse is
// added to the model space block Table Record.
//
void
projectEllipse(AcDbObjectId ellipseId)
{
    AcDbEllipse *pEllipse;
    acdbOpenObject(pEllipse, ellipseId, AcDb::kForRead);

    // Now project the ellipse onto a plane with a
    // normal of <1, 1, 1>.
    //
    AcDbEllipse *pProjectedCurve;
    pEllipse->getOrthoProjectedCurve(AcGePlane(
        AcGePoint3d::kOrigin, AcGeVector3d(1, 1, 1)),
        (AcDbCurve*&)pProjectedCurve);
    pEllipse->close();

    AcDbObjectId newCurveId;
    addToModelSpace(newCurveId, pProjectedCurve);
}

// Accepts an ellipse object ID, opens the ellipse, and uses
// its getOffsetCurves() member function to create a new
// ellipse that is offset 0.5 drawing units from the
// original ellipse.
//
void
offsetEllipse(AcDbObjectId ellipseId)
{
    AcDbEllipse *pEllipse;
    acdbOpenObject(pEllipse, ellipseId, AcDb::kForRead);

    // Now generate an ellipse offset by 0.5 drawing units.
    //
    AcDbVoidPtrArray curves;
    pEllipse->getOffsetCurves(0.5, curves);
    pEllipse->close();

    AcDbObjectId newCurveId;
    addToModelSpace(newCurveId, (AcDbEntity*)curves[0]);
}
```

Associating Hyperlinks with Entities

ObjectARX allows you to associate hyperlinks with entities, by using the classes `AcDbHyperlink`, `AcDbHyperlinkCollection`, and `AcDbEntityHyperlinkPE`. A hyperlink can be a URL or a non-Web address such as a local file. You can attach, view, edit, and list hyperlinks within your application.

An overview of the hyperlink classes follows, but for complete information on the classes and their methods, see the *ObjectARX Reference*.

AcDbHyperlink Class

An `AcDbHyperlink` object contains the hyperlink name (for example, *http://www.autodesk.com*), a sublocation within that link, the hyperlink description or friendly name ("Click here for Autodesk"), and a display string for the hyperlink. For AutoCAD, a sublocation is a named view, while in a spreadsheet application, for example, a sublocation might be a cell or group of cells. The display string is usually the same as the hyperlink's description. If the description is null, the hyperlink's name and sublocation are used instead, in "name – sublocation" format.

Hyperlinks may also have nesting levels. Nesting level is only of interest when dealing with hyperlink collections associated with an entity within a block, or with collections associated with an INSERT entity.

AcDbHyperlinkCollection Class

This class is a collection of `AcDbHyperlink` objects, and has a variety of methods for adding and removing those objects. The `AcDbHyperlinkCollection` deletes its contents when they are removed, and when the collection object itself is deleted. Hyperlinks in the collection are numbered from zero.

AcDbEntityHyperlinkPE Class

The methods of the `AcDbEntityHyperlinkPE` class allow you to set, get, and count the hyperlinks associated with an entity.

Hyperlink Example

The following function lists the hyperlinks associated with an entity and allows new hyperlinks to be added in their place. (Error checking is not shown.)

```
void AddHyperlink()
{
    ads_name en;
    ads_point pt;
    AcDbEntity * pEnt;
    AcDbObjectId pEntId;

    // Prompt user to select entity.
    acedEntSel("\nSelect an Entity:  ", en, pt);

    // Get Object id.
    acdbGetObjectId(pEntId, en);

    // Open object for write.
    acdbOpenObject(pEnt, pEntId, AcDb::kForWrite);

    // The hyperlink collection object is created inside
    // of getHyperlinkCollection
    // below. It is our responsibility to delete it.
    AcDbHyperlinkCollection * pcHCL = NULL;

    // Get the hyperlink collection associated with the entity.
    ACRX_X_CALL(pEnt, AcDbEntityHyperlinkPE)->
        getHyperlinkCollection(pEnt, pcHCL, false, true);

    // If a hyperlink exists already, say so.
    if (pcHCL->count() != 0)
    {
        AcDbHyperlink * pcHO;

        acutPrintf("\nThe following hyperlink info already exists
            on this entity:");

       // Iterate through collection and print existing hyperlinks.
        int i = 0;
        for (i = 0; i < pcHCL->count(); i++)
        {
            // Get point to current hyperlink object.
            pcHO = pcHCL->item(i);

            acutPrintf("\nHyperlink name: %s", pcHO->name());
            acutPrintf("\nHyperlink location: %s",
                pcHO->subLocation());
            acutPrintf("\nHyperlink description: %s",
                pcHO->description());
        }
        acutPrintf("\n** All will be replaced.**");
```

```
        // Remove existing hyperlinks from collection.
        // RemoveAt will delete objects too.
        for (i = pcHCL->count() - 1; i >= 0; i--)
        {
            pcHCL->removeAt(i);
        }
    }

    // Get new hyperlinks for this entity.
    for (;;)
    {
       acutPrintf("\nEnter null name, location, and description to
            terminate input requests.");

        // Prompt user for name and description.
        char sName[100], sLocation[100], sDescription[100];
        if (acedGetString(TRUE, "\nEnter hyperlink name: ", sName)
            != RTNORM)
            acutPrintf("Invalid input\n");
        if (acedGetString(TRUE, "\nEnter hyperlink location: ",
            sLocation) != RTNORM)
            acutPrintf("Invalid input\n");
        if (acedGetString(TRUE, "\nEnter hyperlink description: ",
            sDescription) != RTNORM)
            acutPrintf("Invalid input\n");

        // Add hyperlink or exit prompting.
        if (strcmp(sName, "") || strcmp(sLocation, "") ||
           strcmp(sDescription, ""))
           pcHCL->addTail(sName, sDescription, sLocation);
        else
           break;
    }

    // Add these hyperlinks to the selected entity (opened above).
    ACRX_X_CALL(pEnt, AcDbEntityHyperlinkPE)->
        setHyperlinkCollection(pEnt, pcHCL);

    // Delete the collection. The collection will delete all its
    // contained hyperlink objects.
    delete pcHCL;

    // Close the object.
    pEnt->close();
}
```

Container Objects

In This Chapter

7

- Comparison of Symbol Tables and Dictionaries
- Symbol Tables
- Dictionaries
- Layouts
- Xrecords

This chapter describes the container objects used in AutoCAD database operations: symbol tables, dictionaries, groups, and xrecords. As part of any drawing, AutoCAD creates a fixed set of symbol tables and the named object dictionary, which contains two other dictionaries, the MLINE style and GROUP dictionaries. The chapter examples demonstrate how to add entries to symbol tables, dictionaries, and groups, and how to query the contents of these containers using iterators. They also show how to create and use your own dictionaries and xrecords to manage application data and objects. For a description of the extension dictionary of an `AcDbObject` object, see chapter 5, "Database Objects."

Comparison of Symbol Tables and Dictionaries

Symbol tables and dictionaries perform essentially the same function; they contain entries that are database objects that can be looked up using a text string key. You can add entries to these container objects, and you can use an iterator to step through the entries and query their contents.

The AutoCAD database always contains a fixed set of nine symbol tables, described in the following section. You cannot create or delete a symbol table, but you can add or change the entries in a symbol table, which are called records. Each symbol table contains only a particular type of object. For example, the `AcDbLayerTable` contains only objects of type `AcDbLayerTableRecord`. Symbol tables are defined in this manner mainly for compatibility with AutoCAD Release 12 and previous releases of AutoCAD.

Dictionaries provide a similar mechanism for storing and retrieving objects with associated name keys. The AutoCAD database creates the named object dictionary whenever it creates a new drawing. The named object dictionary can be viewed as the master "table of contents" for the nonentity object structures in a drawing. This dictionary, by default, contains four dictionaries: the GROUP dictionary, the MLINE style dictionary, the layout dictionary, and the plot style name dictionary. You can create any number of additional objects and add them to the named object dictionary. However, the best practice is to add one object directly to the named object dictionary and have that object in turn own the other objects associated with your application. Typically, the owning object is a container class such as a dictionary. Use your assigned four-letter Registered Developer Symbol for the name of this class.

An `AcDbDictionary` object can contain any type of `AcDbObject`, including other dictionaries. A dictionary object does not perform type checking of entries. However, the MLINE style dictionary should contain only instances of class `AcDbMlineStyle`, and the GROUP dictionary should contain only instances of `AcDbGroup`. An application may require specific typing for entries in a dictionary that it creates and maintains.

The class hierarchy for symbol tables, symbol table records, dictionaries, and iterators is as follows.

An important difference between symbol tables and dictionaries is that symbol table records cannot be erased directly by an ObjectARX application. These records can be erased only with the PURGE command or selectively filtered out with wblock operations. Objects owned by a dictionary can be erased.

WARNING! Erasing dictionaries or dictionary entries (see "Essential Database Objects" on page 22) probably will cause AutoCAD or other applications to fail.

Another important difference is that symbol table records store their associated look-up name in a field in their class definition. Dictionaries, on the other hand, store the name key as part of the dictionary, independent of the object it is associated with, as shown in the following figure.

Symbol Tables

Names used in symbol table records and in dictionaries must follow these rules:

- Names can be any length in ObjectARX, but symbol names entered by users in AutoCAD are limited to 255 characters.
- AutoCAD preserves the case of names but does not use the case in comparisons. For example, AutoCAD considers "Floor" to be the same symbol as "FLOOR."
- Names can be composed of all characters allowed in Windows NT filenames, except comma (,), backquote (‘), semi-colon (;), and equal sign (=).

The AutoCAD database contains the following symbol tables (parentheses indicate class name and AutoCAD command used for adding entries):

- Block table (`AcDbBlockTable`; BLOCK)
- Layer table (`AcDbLayerTable`; LAYER)
- Text style table (`AcDbTextStyleTable`; STYLE)
- Linetype table (`AcDbLinetypeTable`; LTYPE)
- View table (`AcDbViewTable`; VIEW)
- UCS table (`AcDbUCSTable`; UCS)
- Viewport table (`AcDbViewportTable`; VPORT)
- Registered applications table (`AcDbRegAppTable`)
- Dimension styles table (`AcDbDimStyleTable`; DIMSTYLE)

Each table contains objects of a corresponding subclass of `AcDbSymbolTableRecord`.

Each symbol table class provides a `getAt()` function for looking up the record specified by name. The signatures for overloaded forms of the `getAt()` function are as follows. (`##BASE_NAME##` stands for any of the nine symbol table class types.)

```
Acad::ErrorStatus
AcDb##BASE_NAME##Table::getAt(const char* pEntryName,
                    AcDb::OpenMode mode,
                    AcDb##BASE_NAME##TableRecord*&
                        pRecord,
                    Adesk::Boolean openErasedRecord =
                        Adesk::kFalse) const;
```

or

```
Acad::ErrorStatus
AcDb##BASE_NAME##Table::getAt(const char* pEntryName,
                    AcDbObjectId& recordId,
                    Adesk::Boolean getErasedRecord =
                        Adesk::kFalse) const;
```

This first version of this function returns a pointer to the opened record in `pRecord` if a matching record is found and the open operation (with the specified mode) succeeds. If `openErasedRecord` is `kTrue`, the function returns the object even if it was erased. If `openErasedRecord` is `kFalse`, the function returns a `NULL` pointer and an error status of `eWasErased` for erased objects.

The second version of the `getAt()` function returns the `AcDbObjectId` of the record specified by name in the value `recordId` if a matching record is found. If `getErasedRecord` is `kTrue`, the function returns the matching object even if it has been erased. The object is not opened.

Once you have obtained a record and opened it, you can get and set different member values. For the specific symbol table record class for a complete list of the class member functions, see the *ObjectARX Reference*.

Other important functions provided by all symbol table classes are the `has()` and `add()` functions. See the example in "Creating and Modifying a Layer Table Record" on page 150. The signature for the `has()` function is

```
Adesk::Boolean
AcDb##BASE_NAME##Table::has(const char* pName) const;
```

The `has()` function returns `kTrue` if the table contains a record with a name that matches `pName`.

The `add()` function has the following signatures:

```
Acad::ErrorStatus
AcDb##BASE_NAME##Table::add(AcDb##BASE_NAME##TableRecord*
                            pRecord);

Acad::ErrorStatus
AcDb##BASE_NAME##Table::add(AcDbObjectId& recordId,
                            AcDb##BASE_NAME##TableRecord*
                            pRecord);
```

This function adds the record pointed to by `pRecord` to both the database containing the table and the table itself. If the additions succeed and the argument `pId` is non-`NULL`, it is set to the `AcDbObjectId` of the record in the database.

Block Table

Entities in the database typically belong to a block table record. The block table contains three records by default, *MODEL_SPACE, *PAPER_SPACE, and *PAPER_SPACE0, which correspond to the three initial drawing spaces that can be edited directly by AutoCAD users. For examples of adding entities to the model space block table record, see chapter 2, "Database Primer," and chapter 6, "Entities."

The *PAPER_SPACE and *PAPER_SPACE0 records correspond to the two predefined paper space layouts in AutoCAD. You can add, modify, and delete paper space layouts.

New block table records are created when the user issues a BLOCK command or an INSERT command to insert an external drawing. New block table records are also created with the `acdbEntMake()` function. The BLOCK? command lists the contents of the block table, with the exception of the *MODEL_SPACE and *PAPER_SPACE records. See chapter 6, "Entities," for examples of block table record and block reference creation. (A block reference is an entity that refers to a given block table record.)

Layer Table

The layer table contains one layer, layer 0, by default. A user adds layers to this table with the LAYER command.

Layer Properties

The `AcDbLayerTableRecord` class contains member functions for specifying a number of layer properties that affect the display of their associated entities. All entities must refer to a valid layer table record. The *AutoCAD User's Guide* provides a detailed description of layer properties.

The following sections list the member functions for setting and querying layer properties.

Frozen/Thawed

When a layer is frozen, graphics are not regenerated.

```
void AcDbLayerTableRecord::setIsFrozen(Adesk::Boolean);

Adesk::Boolean
AcDbLayerTableRecord::isFrozen() const;
```

On/Off
When a layer is OFF, graphics are not displayed.

```
void AcDbLayerTableRecord::setIsOff(Adesk::Boolean);

Adesk::Boolean
AcDbLayerTableRecord::isOff() const;
```

Viewport
This `setVPDFLT()` function specifies whether the layer by default is visible or invisible in new viewports.

```
void AcDbLayerTableRecord::setVPDFLT(Adesk::Boolean);

Adesk::Boolean
AcDbLayerTableRecord::VPDFLT() const;
```

Locked/Unlocked
Entities on a locked layer cannot be modified by an AutoCAD user or opened for the `write()` function within a program.

```
void AcDbLayerTableRecord::setIsLocked(Adesk::Boolean);

Adesk::Boolean
AcDbLayerTableRecord::isLocked() const;
```

Color
The color set by the `setColor()` function is used when an entity's color is BYLAYER.

```
void AcDbLayerTableRecord::setColor(const AcCmColor &color);

AcCmColor
AcDbLayerTableRecord::color() const;
```

Linetype
The linetype set by the `setLinetypeObjectId()` function is used when an entity's linetype is BYLAYER.

```
void AcDbLayerTableRecord::setLinetypeObjectId(AcDbObjectId);

AcDbObjectId
AcDbLayerTableRecord::linetypeObjectId() const;
```

Creating and Modifying a Layer Table Record

The following example shows obtaining the layer table for the current database and opening it for writing. It creates a new layer table record (`AcDbLayerTableRecord`) and sets certain attributes of the layer (name, frozen attribute, on/off, viewport, and locked). Then it creates a color class object and sets the color of the layer to red.

To set the linetype for the layer, this example opens the linetype table for reading and obtains the object ID of the linetype record for the desired line-

type (here, "DASHED"). Once it has the object ID for the linetype, it closes the linetype table and sets the linetype for the new layer table record. This example uses the `add()` function to add the layer table record to the layer table. Finally, it closes the layer table record and the layer table itself.

```
void
addLayer()
{
    AcDbLayerTable *pLayerTbl;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pLayerTbl, AcDb::kForWrite);
    if (!pLayerTbl->has("ASDK_TESTLAYER")) {
        AcDbLayerTableRecord *pLayerTblRcd
            = new AcDbLayerTableRecord;
        pLayerTblRcd->setName("ASDK_TESTLAYER");
        pLayerTblRcd->setIsFrozen(0);// layer to THAWED
        pLayerTblRcd->setIsOff(0);   // layer to ON
        pLayerTblRcd->setVPDFLT(0);  // viewport default
        pLayerTblRcd->setIsLocked(0);// un-locked

        AcCmColor color;
        color.setColorIndex(1); // set color to red
        pLayerTblRcd->setColor(color);

        // For linetype, we need to provide the object ID of
        // the linetype record for the linetype we want to
        // use.  First, we need to get the object ID.
        //
        AcDbLinetypeTable *pLinetypeTbl;
        AcDbObjectId ltId;
        acdbHostApplicationServices()->workingDatabase()
            ->getSymbolTable(pLinetypeTbl, AcDb::kForRead);
        if ((pLinetypeTbl->getAt("DASHED", ltId))
            != Acad::eOk)
        {
            acutPrintf("\nUnable to find DASHED"
                " linetype. Using CONTINUOUS");

            // CONTINUOUS is in every drawing, so use it.
            //
            pLinetypeTbl->getAt("CONTINUOUS", ltId);
        }
        pLinetypeTbl->close();

        pLayerTblRcd->setLinetypeObjectId(ltId);
        pLayerTbl->add(pLayerTblRcd);
        pLayerTblRcd->close();
        pLayerTbl->close();
    } else {
        pLayerTbl->close();
        acutPrintf("\nlayer already exists");
    }
}
```

Iterators

Each symbol table has a corresponding iterator that you can create with the `AcDb##BASE_NAME##Table::newIterator()` function.

```
Acad::ErrorStatus
AcDb##BASE_NAME##Table::newIterator(
    AcDb##BASE_NAME##TableIterator*& pIterator,
    Adesk::Boolean atBeginning = Adesk::kTrue,
    Adesk::Boolean skipErased = Adesk::kTrue) const;
```

The `newIterator()` function creates an object that can be used to step through the contents of the table and sets `pIterator` to point to the iterator object. If `atBeginning` is `kTrue`, the iterator starts at the beginning of the table; if `kFalse`, it starts at the end of the table. If the `skipErased` argument is `kTrue`, the iterator is positioned initially at the first (or last) unerased record; if `kFalse`, it is positioned at the first (or last) record, regardless of whether it has been erased. For a description of the functions available for each iterator class, see the *ObjectARX Reference*.

When you create a new iterator, you are also responsible for deleting it. A symbol table should not be closed until all of the iterators it has constructed have been deleted.

In addition to the symbol tables, the block table record has an iterator that operates on the entities it owns. The `AcDbBlockTableRecord` class returns an object of class `AcDbBlockTableRecordIterator` when you ask it for a new iterator. This iterator enables you to step through the entities contained in the block table record and to seek particular entities.

Iterating over Tables

The code in the following example creates an iterator that walks through the symbol table records in the linetype table. It obtains each record, opens it for read, obtains the linetype name, closes the record, and then prints the linetype name. At the end, the program deletes the iterator.

```
void
iterateLinetypes()
{
    AcDbLinetypeTable *pLinetypeTbl;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pLinetypeTbl, AcDb::kForRead);

    // Create a new iterator that starts at table
    // beginning and skips deleted.
    //
    AcDbLinetypeTableIterator *pLtIterator;
    pLinetypeTbl->newIterator(pLtIterator);
```

```
    // Walk the table, getting every table record and
    // printing the linetype name.
    //
    AcDbLinetypeTableRecord *pLtTableRcd;
    char *pLtName;
    for (; !pLtIterator->done(); pLtIterator->step()) {
        pLtIterator->getRecord(pLtTableRcd, AcDb::kForRead);
        pLtTableRcd->getName(pLtName);
        pLtTableRcd->close();
        acutPrintf("\nLinetype name is:  %s", pLtName);
        free(pLtName);
    }
    delete pLtIterator;
    pLinetypeTbl->close();
}
```

Dictionaries

To create a new dictionary, you need to create an instance of `AcDbDictionary`, add it to the database, and register it with its owner object. Use the `setAt()` function of `AcDbDictionary` to add objects to the dictionary and the database. The signature of this function is

```
Acad::ErrorStatus
AcDbDictionary::setAt(const char*   pSrchKey,
                      AcDbObject*   pNewValue,
                      AcDbObjectId& retObjId);
```

The `setAt()` function adds a new entry specified by `newValue` to the dictionary. If the entry already exists, it is replaced by the new value. The name of the object is specified by `srchKey`. The object ID of the entry is returned in `retObjId`.

When you add an entry to a dictionary, the dictionary automatically attaches a reactor to the entry. If the object is erased, the dictionary is notified and removes it from the dictionary.

Groups and the Group Dictionary

A group is a container object that maintains an ordered collection of database entities. Groups can be thought of as named persistent selection sets. They do not have an ownership link to the entities they contain.

When an entity is erased, it is automatically removed from the groups that contain it. If an entity is unerased, it is automatically reinserted into the group.

Use the `AcDbGroup::newIterator()` function to obtain an iterator and step through the entities in the group. The `AcDbGroup` class also provides functions for appending and prepending entities to the group, inserting entities at a particular index in the group, removing entities, and transferring entities from one position in the group to another. See `AcDbGroup` in the *ObjectARX Reference*.

You can also assign properties to all members of a group using the `setColor()`, `setLayer()`, `setLinetype()`, `setVisibility()`, and `setHighlight()` functions of the `AcDbGroup` class. These operations have the same effect as opening each entity in the group and setting its property directly.

Groups should always be stored in the GROUP dictionary, which can be obtained as follows:

```
AcDbDictionary* pGrpDict =
    acdbHostApplicationServices()->working Database()->
        getGroupDictionary(pGroupDict, AcDb::kForWrite);
```

An alternative way to obtain the GROUP dictionary is to look up "ACAD_GROUP" in the named object dictionary.

The following functions are part of an application that first prompts the user to select some entities that are placed into a group called "ASDK_GROUPTEST". Then it calls the function `removeAllButLines()` to iterate over the group and remove all the entities that are not lines. Finally, it changes the remaining entities in the group to red.

```
void
groups()
{
    AcDbGroup *pGroup = new AcDbGroup("grouptest");

    AcDbDictionary *pGroupDict;
    acdbHostApplicationServices()->workingDatabase()
        ->getGroupDictionary(pGroupDict, AcDb::kForWrite);

    AcDbObjectId groupId;
    pGroupDict->setAt("ASDK_GROUPTEST", pGroup, groupId);
    pGroupDict->close();
    pGroup->close();

    makeGroup(groupId);
    removeAllButLines(groupId);
}

// Prompts the user to select objects to add to the group,
// opens the group identified by "groupId" passed in as
// an argument, then adds the selected objects to the group.
//
```

```
void
makeGroup(AcDbObjectId groupId)
{
    ads_name sset;
    int err = acedSSGet(NULL, NULL, NULL, NULL, sset);

    if (err != RTNORM) {
        return;
    }
    AcDbGroup *pGroup;
    acdbOpenObject(pGroup, groupId, AcDb::kForWrite);

    // Traverse the selection set, exchanging each ads_name
    // for an object ID, then adding the object to the group.
    //
    long i, length;
    ads_name ename;
    AcDbObjectId entId;
    acedSSLength(sset, &length);
    for (i = 0; i < length; i++) {
        acedSSName(sset, i, ename);
        acdbGetObjectId(entId, ename);
        pGroup->append(entId);
    }
    pGroup->close();
    acedSSFree(sset);
}

// Accepts an object ID of an AcDbGroup object, opens it,
// then iterates over the group, removing all entities that
// are not AcDbLines and changing all remaining entities in
// the group to color red.
//
void
removeAllButLines(AcDbObjectId groupId)
{
    AcDbGroup *pGroup;

    acdbOpenObject(pGroup, groupId, AcDb::kForWrite);
    AcDbGroupIterator *pIter = pGroup->newIterator();
    AcDbObject *pObj;
    for (; !pIter->done(); pIter->next()) {
        pIter->getObject(pObj, AcDb::kForRead);

        // If it is not a line or descended from a line,
        // close it and remove it from the group.  Otherwise,
        // just close it.
        //
```

```
        if (!pObj->isKindOf(AcDbLine::desc())) {
            // AcDbGroup::remove() requires that the object
            // to be removed be closed, so close it now.
            //
            pObj->close();
            pGroup->remove(pIter->objectId());
        } else {
            pObj->close();
        }
    }

    delete pIter;

    // Now change the color of all the entities in the group
    // to red (AutoCAD color index number 1).
    //
    pGroup->setColorIndex(1);
    pGroup->close();
}
```

MLINE Style Dictionary

The MLINE style dictionary contains objects of class `AcDbMlineStyle`. As shown in the following figure, objects of class `AcDbMline` each have an associated MLINE style that specifies the properties of the multiline, such as offset, color, and linetype.

Layout Dictionary

The layout dictionary is a default dictionary within the named object dictionary that contains objects of class `AcDbLayout`. The `AcDbLayout` object stores the characteristics of a paper space layout, including the plot settings. Each `AcDbLayout` object also contains an object ID of an associated block table record, which stores the entities associated with the layout.

Creating a Dictionary

The following example creates a new dictionary (ASDK_DICT) and adds it to the named object dictionary. Then it creates two new objects of the custom class `AsdkMyClass` (derived from `AcDbObject`) and adds them to the dictionary using the `setAt()` function.

NOTE You need to close the objects after adding them with the `setAt()` function.

```
// This function creates two objects of class AsdkMyClass.
// It fills them in with the integers 1 and 2, and then adds
// them to the dictionary associated with the key ASDK_DICT. If this
// dictionary doesn't exist, it is created and added to the named
// object dictionary.
//
void
createDictionary()
{
    AcDbDictionary *pNamedobj;
    acdbHostApplicationServices()->workingDatabase()->
        getNamedObjectsDictionary(pNamedobj, AcDb::kForWrite);

    // Check to see if the dictionary we want to create is
    // already present. If not, create it and add
    // it to the named object dictionary.
    //
```

```
    AcDbDictionary *pDict;
    if (pNamedobj->getAt("ASDK_DICT", (AcDbObject*&) pDict,
        AcDb::kForWrite) == Acad::eKeyNotFound)
    {
        pDict = new AcDbDictionary;
        AcDbObjectId DictId;
        pNamedobj->setAt("ASDK_DICT", pDict, DictId);
    }

    pNamedobj->close();
    if (pDict) {
        // Create new objects to add to the new dictionary,
        // add them, then close them.
        //
        AsdkMyClass *pObj1 = new AsdkMyClass(1);
        AsdkMyClass *pObj2 = new AsdkMyClass(2);

        AcDbObjectId rId1, rId2;
        pDict->setAt("OBJ1", pObj1, rId1);
        pDict->setAt("OBJ2", pObj2, rId2);

        pObj1->close();
        pObj2->close();
        pDict->close();
    }
}
```

Iterating over Dictionary Entries

The iterator class for dictionaries is `AcDbDictionaryIterator`. The following code excerpt obtains a dictionary (ASDK_DICT) from the named object dictionary. It then uses a dictionary iterator to step through the dictionary entries and print the value of the stored integer. Finally, it deletes the iterator and closes the dictionary.

```
void
iterateDictionary()
{
    AcDbDictionary *pNamedobj;
    acdbHostApplicationServices()->workingDatabase()
        ->getNamedObjectsDictionary(pNamedobj, AcDb::kForRead);

    // Get a pointer to the ASDK_DICT dictionary.
    //
    AcDbDictionary *pDict;
    pNamedobj->getAt("ASDK_DICT", (AcDbObject*&)pDict,
        AcDb::kForRead);
    pNamedobj->close();

    // Get an iterator for the ASDK_DICT dictionary.
    //
    AcDbDictionaryIterator* pDictIter = pDict->newIterator();
    AsdkMyClass *pMyCl;
    Adesk::Int16 val;
```

```
    for (; !pDictIter->done(); pDictIter->next()) {
        // Get the current record, open it for read, and
        // print its data.
        //
        pDictIter->getObject((AcDbObject*&)pMyCl,
            AcDb::kForRead);
        pMyCl->getData(val);
        pMyCl->close();
        acutPrintf("\nintval is:  %d", val);
    }
    delete pDictIter;
    pDict->close();
}
```

Layouts

AutoCAD initially contains three layouts: a model space layout and two paper space layouts. These layouts can be accessed by tabs at the bottom of the drawing window in AutoCAD. The tabs are initially named Model, Layout1, and Layout2.

The Model tab is the default tab and represents model space, in which you generally create your drawing. The Layout1 and Layout2 tabs represent paper space and are generally used for laying out your drawing for printing. The paper space layouts display a paper image that shows the printable boundary for the configured print device.

It is recommended that you use paper space layouts for preparing final drawings for output, but printing can be performed from any layout, including the model space layout. For more information on using layouts in AutoCAD, see the *AutoCAD User's Guide*.

ObjectARX Layout Classes

The main classes involved in creating and manipulating layouts are the following:

- `AcDbLayout`
- `AcDbPlotSettings`
- `AcDbPlotSettingsValidator`
- `AcDbLayoutManager`
- `AcApLayoutManager`
- `AcDbLayoutManagerReactor`

`AcDbLayout`, `AcDbPlotSettings`, and `AcDbPlotSettingsValidator` are used to create and set attributes on layout objects. `AcDbLayoutManager`, `AcApLayoutManager`, and `AcDbLayoutManagerReactor` are used to manipulate

layout objects and to perform other layout-related tasks. The following sections provide an overview of some of these classes. For more information, see the *ObjectARX Reference*. For an example of using the ObjectARX layout classes, see the *lmgrtest.arx* sample application in the ObjectARX *samples* directory.

Layout Objects

Information about layouts is stored in instances of the `AcDbLayout` class. A layout object contains the printing and plotting settings information needed to print the desired portion of the drawing. For example, a layout object contains the plot device, media size, plot area, and plot rotation, as well as several other attributes that help define the area to be printed.

`AcDbLayout` objects are stored in the ACAD_LAYOUT dictionary within the named object dictionary of the database. There is one `AcDbLayout` object per paper space layout, as well as a single `AcDbLayout` for model space. Each `AcDbLayout` object contains the object ID of its associated `AcDbBlockTableRecord`. This makes it easy to find the block table record in which the layout's actual geometry resides. If an `AcDbBlockTableRecord` represents a layout, then it contains the object ID of its associated `AcDbLayout` object.

Most of the plot information for layout objects is stored in `AcDbPlotSettings`, the base class of `AcDbLayout`. You can create named plot settings and use them to initialize other `AcDbLayout` objects. This allows you to export and import plot settings from one layout to another. These named plot settings are stored in instances of the `AcDbPlotSettings` class. There is one `AcDbPlotSettings` object for each named plot setting and they are stored in the ACAD_PLOTSETTINGS dictionary within the named object dictionary.

NOTE There is no direct connection between `AcDbLayout` objects in the ACAD_LAYOUT dictionary and `AcDbPlotSettings` objects in the ACAD_PLOTSETTINGS dictionary.

The Layout Manager

You can manage `AcDbLayout` objects by using the `AcApLayoutManager` class. The `AcApLayoutManager` class allows you to

- Create layouts
- Delete layouts
- Rename layouts
- Copy and clone layouts

- Set the current layout
- Find a particular layout
- Set the plot characteristics of a layout

There is one instance of a layout manager per application. The layout manager always operates on the current layout.

Xrecords

Xrecords enable you to add arbitrary, application-specific data. Because they are an alternative to defining your own object class, they are especially useful to AutoLISP programmers. An xrecord is an instance of class `AcDbxrecord`, which is a subclass of `AcDbObject`. Xrecord state is defined as the contents of a resbuf chain, which is a list of data groups, each of which in turn contains a DXF group code plus associated data. The value of the group code defines the associated data type. Group codes for xrecords are in the range of 1 through 369. The following section describes the available DXF group codes.

There is no inherent size limit to the amount of data you can store in an xrecord. Xrecords can be owned by any other object, including the extension dictionary of any object, the named object dictionary, any other dictionary, or other xrecords.

No notification is sent when an xrecord is modified. If an application needs to know when an object owning an xrecord has been modified, the application will need to send its own notification.

The `AcDbXrecord` class provides two member functions for setting and obtaining resbuf chains, the `setfromRbChain()` and `rbChain()` functions:

```
Acad::ErrorStatus
AcDbXrecord::setFromRbChain(
    resbuf& pRb,
    AcDbDatabase* auxDb=NULL);

Acad::ErrorStatus
AcDbXrecord::rbChain(
    resbuf** ppRb,
    AcDbDatabase* auxDb=NULL) const;
```

The `AcDbXrecord::setFromRbChain()` function replaces the existing resbuf chain with the chain passed in.

DXF Group Codes for Xrecords

The following table lists the DXF group codes that can be used in xrecords.

DXF group code ranges for xrecords		
From	**To**	**Data Type**
1	4	Text
6	9	Text
10	17	Point or vector (3 reals)
38	59	Real
60	79	16-bit integer
90	99	32-bit integer
102	102	Control string “{“ or “}”
140	149	real
170	179	16-bit integer
210	219	Real
270	279	16-bit integer
280	289	8-bit integer
300	309	Text
310	319	Binary chunk
320	329	Handle
330	339	Soft pointer ID
340	349	Hard pointer ID
350	359	Soft ownership ID
360	369	Hard ownership ID

For a description of hard and soft owners and pointers, see chapter 12, “Deriving from AcDbObject.”

Examples

The following ObjectARX examples consist of two functions: `createXrecord()` and `listXrecord()`. The first function adds a new xrecord to a dictionary, adds the dictionary to the named object dictionary, and then adds data to the xrecord. The `listXrecord()` function opens an xrecord, obtains its data list, and sends the list to be printed. For the complete program, see the samples directory.

```
void
createXrecord()
{
    AcDbDictionary *pNamedobj, *pDict;
    acdbHostApplicationServices()->workingDatabase()
        ->getNamedObjectsDictionary(pNamedobj, AcDb::kForWrite);

    // Check to see if the dictionary we want to create is
    // already present. If not, then create it and add
    // it to the named object dictionary.
    //
    if (pNamedobj->getAt("ASDK_DICT", (AcDbObject*&) pDict,
        AcDb::kForWrite) == Acad::eKeyNotFound)
    {
        pDict = new AcDbDictionary;
        AcDbObjectId DictId;
        pNamedobj->setAt("ASDK_DICT", pDict, DictId);
    }
    pNamedobj->close();

    // Add a new xrecord to the ASDK_DICT dictionary.
    //
    AcDbXrecord *pXrec = new AcDbXrecord;
    AcDbObjectId xrecObjId;
    pDict->setAt("XREC1", pXrec, xrecObjId);
    pDict->close();

    // Create a resbuf list to add to the xrecord.
    //
    struct resbuf *pHead;
    ads_point testpt = {1.0, 2.0, 0.0};
    pHead = acutBuildList(AcDb::kDxfText,
        "This is a test Xrecord list",
        AcDb::kDxfXCoord, testpt,
        AcDb::kDxfReal, 3.14159,
        AcDb::kDxfAngle, 3.14159,
        AcDb::kDxfColor, 1,
        AcDb::kDxfInt16, 180,
        0);

    // Add the data list to the xrecord.  Notice that this
    // member function takes a reference to resbuf, NOT a
    // pointer to resbuf, so you must dereference the
    // pointer before sending it.
    //
```

```
    pXrec->setFromRbChain(*pHead);
    acutRelRb(pHead);
    pXrec->close();
}

// Gets the xrecord associated with the key XREC1 and
// lists out its contents by passing the resbuf list to the
// function printList.
//
void
listXrecord()
{
    AcDbDictionary *pNamedobj;
    acdbHostApplicationServices()->workingDatabase()
        ->getNamedObjectsDictionary(pNamedobj, AcDb::kForRead);

    // Get the dictionary object associated with the key ASDK_DICT.
    //
    AcDbDictionary *pDict;
    pNamedobj->getAt("ASDK_DICT", (AcDbObject*&)pDict,
        AcDb::kForRead);
    pNamedobj->close();

    // Get the xrecord associated with the key XREC1.
    //
    AcDbXrecord *pXrec;
    pDict->getAt("XREC1", (AcDbObject*&) pXrec,
        AcDb::kForRead);
    pDict->close();

    struct resbuf *pRbList;
    pXrec->rbChain(&pRbList);
    pXrec->close();

    printList(pRbList);
    acutRelRb(pRbList);
}
```

Part II

User Interfaces

MFC Topics

In This Chapter

8

- Introduction
- Using MFC with ObjectARX Applications
- ObjectARX Applications with Dynamically Linked MFC
- Built-In MFC User Interface Support
- Using AdUi and AcUi with VC++ AppWizard

The Microsoft Foundation Class (MFC) library allows a developer to implement standard user interfaces quickly. The ObjectARX environment provides a set of classes that a developer can use to create MFC-based user interfaces that behave and appear as the built-in Autodesk user interfaces. This chapter describes how to use the MFC library as part of an ObjectARX application.

Introduction

ObjectARX applications can be created to take advantage of the Microsoft Foundation Class (MFC) library. This chapter discusses how to build your ObjectARX applications to make use of MFC and how the AutoCAD built-in MFC system can be used to create dialogs that behave and operate like AutoCAD.

Using MFC with ObjectARX Applications

You have the choice of building ObjectARX applications with either a dynamically linked MFC library or a statically linked MFC library. You also have the choice of using a regular DLL or an extension DLL.

NOTE It is highly recommended to dynamically link your MFC ObjectARX application AND make it an extension DLL, since it is the only method that allows you to use the Autodesk AdUi and AcUi MFC base classes.

For complete information about MFC, see the Microsoft online help and technical notes. In particular, see notes 11 and 33 for information about using MFC as part of a DLL, which is an important concept for ObjectARX.

MFC and Modeless Dialog Boxes

Since AutoCAD attempts to take focus away from all of its child windows, modeless dialogs have a special requirement. At regular intervals, the modeless dialog will get a `WM_ACAD_KEEPFOCUS` window message, which is defined in *adscodes.h* as 1001. When your dialog gets this message, it must return `TRUE` if it should keep focus. If the response to this message is `FALSE` (which is also the default), then your dialog box will lose focus as soon as the user moves the mouse pointer off the dialog box's window.

You can do this with the dialog box's message map, and an `ON_MESSAGE()` declaration such as

```
BEGIN_MESSAGE_MAP(HelloDlg, CDialog)
    ON_COMMAND(IDCLOSE, OnClose)
    ON_COMMAND(IDC_DRAW_CIRCLE, OnDrawCircle)
    ON_MESSAGE(WM_ACAD_KEEPFOCUS, onAcadKeepFocus)
END_MESSAGE_MAP()
```

In this example, the application's dialog class is `HelloDlg`, which is derived from `CDialog`. When you add this entry to the message map, you must also

write a handler function for the message. Assume you have written a function called `keepTheFocus()`, which returns `TRUE` if your dialog wants to keep the input focus and `FALSE` if the dialog is willing to yield the focus to AutoCAD. An example message handler is provided here:

```
afx_msg LONG HelloDlg::onAcadKeepFocus(UINT, LONG)
{
    return keepTheFocus() ? TRUE : FALSE;
}
```

ObjectARX Applications with Dynamically Linked MFC

The preferred method for building an MFC-based ObjectARX application is to use the dynamically linked MFC libraries.

Visual C++ Project Settings for Dynamically Linked MFC

To build an ObjectARX application using the shared MFC library

1. Select the MFC AppWizard (DLL) option for the project.
2. Select Extension DLL using shared MFC DLL.
3. Go to the Project Settings dialog box and select the General tab.
4. Select Use MFC in a Shared DLL for the Microsoft Foundation Classes field.
5. Add an `acrxEntryPoint` function to the project's CPP file. See the example at the end of the chapter for a complete setup for an MFC project.

Debugging ObjectARX Applications with Dynamic MFC

When debugging ObjectARX applications built with a dynamically linked MFC library, link with the release version of C runtime and MFC libraries. This allows use of the MFC or C runtime debugging facilities, but does not allow stepping into the Microsoft MFC debugging source code.

Resource Management

Resource management is an important consideration when designing an ObjectARX application that uses an MFC library shared with AutoCAD and other applications.

You must insert your module state (using `CDynaLinkLibrary`) into the chain that MFC examines when it performs operations such as locating a resource. However, it is strongly recommended that you explicitly manage your application's resources so that they will not conflict with other resources from AutoCAD or other ObjectARX applications.

To explicitly set resources

1. Before taking any steps that would cause MFC to look for your resource, call the AFX function `AfxSetResourceHandle()` to set the custom resource as the system default.
2. Before setting the system resource to your resource, call `AfxGetResourceHandle()` to get the current system resource.
3. Immediately after performing any functions that require the custom resource, the system resource should be reset to the resource handle previously saved.

Calling AutoCAD API functions (or invoking AutoCAD commands) inside the dialog command handler that needs AutoCAD's resources, such as `acedGetFileD()`, sets the resource back to AutoCAD before calling the functions. Restore your application resource afterwards. (Use `acedGetAcadResourceInstance()` to get AutoCAD's resource handle.)

CAcExtensionModule Class

The ObjectARX SDK provides two simple C++ classes that can be used to make resource management easier. The `CAcExtensionModule` class serves two purposes—it provides a placeholder for an `AFX_EXTENSION_MODULE` structure (normally used to initialize or terminate an MFC extension DLL) and tracks two resource providers for the DLL. The resource providers are the module's resources (which are normally the DLL itself, but may be set to some other module) and the default resources (normally the host application, but are actually the provider currently active when `AttachInstance()` is called). `CAcExtensionModule` tracks these to simplify switching MFC resource lookup between the default and the module's. A DLL should create one instance of this class and provide the implementation for the class.

CAcModuleResourceOverride Class

Use an instance of this class to switch between resource providers. When the object is constructed, a new resource provider will get switched in. Upon destruction, the original resource provider will be restored. The following code provides an example:

```
void MyFunc ()
{
    CAcModuleResourceOverride myResources;
}
```

Upon entry to this function the module's resources will be selected. When the function returns, the default resources will be restored. A resource override can be used in any of three ways:

- Use the default constructor (no arguments) to switch to the module's resources. The default resources will be restored by the destructor. The module/default resources are those maintained by the DLL's `CAcExtensionModule`.
- Pass `NULL` (or 0) to the constructor. The DLL's resources will be selected and the resources that were in effect will be restored when the override object is destroyed.
- Pass a non-`NULL` handle to the constructor. The associated module's resources will be selected and the resources that were in effect will be restored when the override object is destroyed.

There are two macros provided, called `AC_DECLARE_EXTENSION_MODULE` and `AC_IMPLEMENT_EXTENSION_MODULE`, to help define and implement the classes in your application.

The following code illustrates how to make use of the `CAcExtensionModule` and `CAcModuleResourceOverride` classes in an ObjectARX application:

```
AC_IMPLEMENT_EXTENSION_MODULE(theArxDLL);
HINSTANCE _hdllInstance = NULL;
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    // Remove this if you use lpReserved
    UNREFERENCED_PARAMETER(lpReserved);
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        theArxDLL.AttachInstance(hInstance);
        hdllInstance = hInstance;
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        theArxDLL.DetachInstance();
    }
    return 1;   // ok
}
```

Built-In MFC User Interface Support

ObjectARX has a set of MFC User Interface (UI) related classes that easily allow you to provide a consistent UI. This means your UI can behave like and have the appearance of the AutoCAD UI. It is highly recommended to make use of these classes, since they allow your application to be more tightly integrated with the AutoCAD UI. The Autodesk MFC system is divided into two libraries. The first is called AdUi and is not AutoCAD-specific. The second is called AcUi and contains AutoCAD-specific appearance and behavior.

AdUi is an MFC extension dynamic-link library used to extend some of the UI-related classes of MFC. The library was developed for use with AutoCAD and other Autodesk products and contains core functionality. The companion library, AcUi, builds upon the AdUi framework and provides AutoCAD- specific appearance and behavior. The AdUi and AcUi libraries provide classes that extend those provided by MFC in ways that allow ARX developers to use the same UI functionality found in AutoCAD. MFC developers can seamlessly use these classes. Listed below are the main areas of added functionality provided by AdUi and AcUi.

To use AdUi in an MFC-based application, the project's C++ source files must include *adui.h* and the project should link *adui15.lib* (the *adui15.dll* import library).

To use AcUi in an MFC-based AutoCAD application, the project's C++ source files must include *adui.h*, then *acui.h*, and the project should link *acui15.lib* and *adui15.lib*. AutoCAD invokes the library's initialization routine, `InitAcUiDLL()`, which also handles the AdUi initialization (via an `InitAdUiDLL()` call); therefore your application need not reinitialize AcUi or AdUi.

WARNING! Although *adui15.dll* may be called from MFC-based applications other than AutoCAD (or other Autodesk products), the library's intended use is by Autodesk and third parties expressly for the creation of software to work exclusively with AutoCAD, or other Autodesk products. Use of this DLL for non-AutoCAD, standalone products is not permitted under the AutoCAD license agreement.

AdUi and AcUi provide classes that implement the following features:

- Dialog resizing
- Dialog data persistency
- Tabbed dialogs

- Extensible tabbed dialogs
- Context-sensitive help and F1 help
- Dialog interaction with AutoCAD's drawing editor
- Bitmap buttons that are easy to use
- Static bitmap buttons
- Bitmap buttons that are drag and drop sites
- Toolbar-style bitmap buttons
- Owner-draw buttons that are easy to use
- Dialog and control support for standard ToolTips
- Dialog and control support for TextTips (which display truncated text)
- Dialog and control support for DrawTips (owner-draw TextTips)
- Custom messaging, including data validation
- Combo boxes that display and allow the selection of many AutoCAD specific items
- Docking control bar windows for use with AutoCAD
- AutoCAD-specific bitmap buttons (stock Pick and Select buttons)
- Specialized edit controls that can perform AutoCAD-specific data validation
- Custom messaging, including data validation

Class Hierarchy

The following are the supported classes in the AdUi and AcUi libraries. There are classes present in the header files that are not shown and are for internal use only and are not supported for use with ObjectARX.

```
CDialog
└─ CAdUiBaseDialog
   └─ CAdUiDialog
      ├─ CAdUiTabMainDialog
      │  └─ CAcUiTabMainDialog
      ├─ CAdUiTabChildDialog
      │  └─ CAcUiTabChildDialog
      ├─ CAdUiAlertDialog
      └─ CAcUiDialog

CComboBox
└─ CAdUiComboBox
   └─ CAcUiComboBox
      ├─ CAcUiAngleComboBox
      ├─ CAcUiMRUComboBox
      │  ├─ CAcUiArrowHeadComboBox
      │  ├─ CAcUiColorComboBox
      │  ├─ CAcUiLineWeightComboBox
      │  ├─ CAcUiPlotStyleNamesComboBox
      │  └─ CAcUiPlotStyleTablesComboBox
      ├─ CAcUiNumericComboBox
      ├─ CAcUiStringComboBox
      └─ CAcUiSymbolComboBox

CFileDialog
└─ CAdUiFileDialog
   └─ CAcUiFileDialog

CHeaderCtrl
└─ CAdUiHeaderCtrl

CControlBar
└─ CAdUiDockControlBar
   └─ CAcUiDockControlBar

CEdit
└─ CAdUiEdit
   └─ CAcUiEdit
      ├─ CAcUiAngleEdit
      ├─ CAcUiNumericEdit
      ├─ CAcUiStringEdit
      └─ CAcUiSymbolEdit

CListBox
├─ CAdUiListBox
│  └─ CAcUiListBox
└─ CAcUiMRUListBox

CTabCtrl
└─ CAdUiTab
   └─ CAcUiTab

CAdUiTabExtensionManager

CWnd
└─ CAdUiTipWindow
   └─ CAdUiTextTip

CAdUiDrawTipText

CButton
└─ CAdUiOwnerDrawButton
   └─ CAdUiBitmapButton
      ├─ CAcUiPickButton
      │  └─ CAcUiSelectButton
      ├─ CAdUiBitmapStatic
      │  └─ CAdUiDropSite
      └─ CAdUiToolButton

CListCtrl
└─ CAdUiListCtrl
   └─ CAcUiListCtrl
```

AdUi Messaging

The AdUi library uses an internal messaging scheme to facilitate communication between objects. Typically this involves a container (such as a dialog) responding to a notification from a contained window (such as a control). Advanced applications may tailor the built-in system to their needs, or add AdUi messaging support to other `CWnd` derived classes.

AdUi Tip Windows

AdUi provides three types of tip windows: ToolTips, TextTips, and DrawTips. ToolTips represent stock Windows ToolTips, as provided by the Common Controls DLL installed on the user's system. TextTips are text-based tip windows that pop up over a control, usually to reveal data that the user would otherwise have to scroll into view. DrawTips are an extension of TextTips. The control underneath the tip is usually responsible for painting the contents of the tip (analogous to an owner-draw tip).

Most applications rarely involve these classes directly, since AdUi usually handles all of the requirements. AdUi uses its internal messaging system to negotiate between containers and controls and decide when and how to display a tip.

CAdUiTipWindow Class

`CAdUiTipWindow` is the basic AdUi tip window class. These objects handle generic tip display and know when to automatically hide themselves (such as detecting cursor movement, a brief time-out, or keyboard activity).

CAdUiTextTip Class

`CAdUiTextTip` specializes `CAdUiTipWindow` to display a TextTip.

CAdUiDrawTipText Class

`CAdUiDrawTipText` is used internally by the AdUi messaging system to inform a control that a tip window needs repainting. The control has the option of changing attributes of the tip window's device context and drawing the text.

AdUi Dialog Classes

The AdUi dialog classes are usable in applications other than AutoCAD.

CAdUiBaseDialog Class

`CAdUiBaseDialog` provides basic support for tip windows (ToolTips and TextTips) and the AdUi message handling system. It also supports context

help and F1 help in dialogs. It is the common base class for all dialogs except those based on the common file dialog.

CAdUiDialog Class

`CAdUiDialog` is a general purpose class that provides a set of member functions allowing for resizable dialogs and data persistency.

CAdUiFileDialog

`CAdUiFileDialog` specializes `CFileDialog` much the same way as `CAdUiBaseDialog` specializes `CDialog`. The class provides basic support for tip windows (ToolTips and TextTips), context help and AdUi message handling in a common file dialog. Unlike `CAdUiBaseDialog`, there is no built-in support for position and size persistency.

CAdUiTabMainDialog Class

`CAdUiTabMainDialog` represents the main container dialog in a tabbed dialog. `CAdUiTabMainDialog` and `CAdUiTabMainDialog` are used in place of `CPropertySheet` and `CPropertyPage` to construct tabbed dialogs.

CAdUiTabChildDialog Class

`CAdUiTabChildDialog` represents a tab in a tabbed dialog. `CAdUiTabMainDialog` and `CAdUiTabChildDialog` are used in place of `CPropertySheet` and `CPropertyPage` to construct tabbed dialogs. Each tab in a tabbed dialog is a `CAdUiTabChildDialog`.

AcUi Dialog Classes

The AcUi dialog classes build upon the AdUi dialog classes and are usable only with AutoCAD.

CAcUiDialog Class

`CAcUiDialog` is a general-purpose class that provides a set of member functions allowing for resizable dialogs and data persistency in AutoCAD.

CAcUiTabMainDialog Class

`CAcUiTabMainDialog` represents the main container dialog in an AutoCAD tabbed dialog. `CAcUiTabMainDialog` and `CAcUiTabMainDialog` are used in place of `CPropertySheet` and `CPropertyPage` to construct tabbed dialogs in AutoCAD.

CAcUiTabChildDialog Class

`CAcUiTabChildDialog` represents a tab in a tabbed dialog. `CAcUiTabMainDialog` and `CAcUiTabChildDialog` are used in place of

CPropertySheet and CPropertyPage to construct tabbed dialogs in AutoCAD. Each tab in an AutoCAD tabbed dialog is a CAcUiTabChildDialog.

CAcUiAlertDialog Class

CAdUiAlertDialog represents an alert dialog with three buttons. One button is the CANCEL button and the other two button labels are set by the programmer. It is a general-purpose alert dialog.

CAcUiFileDialog Class

CAcUiFileDialog provides an AutoCAD-specific derivation of CAdUiFileDialog.

AdUi Classes Supporting Tab Extensibility

The following classes provide support for tab dialogs.

CAdUiTabExtensionManager Class

CAdUiDialogManager is a class that manages adding and removing tabs from a tabbed dialog that is extensible. If a dialog is tab extensible, an instance of this class is found in the CAdUiTabMainDialog.

CAdUiTab Class

CAdUiTab encapsulates the MFC CTabCtrl and adds functionality to it. One of these objects is found in the main dialog object.

AdUi and AcUi Control Bar Classes

The following classes provide support for docking windows.

CAdUiDockControlBar Class

The CAdUiDockControlBar class, part of a docking system, adds extended capabilities to the MFC CControlBar class. The main feature provided is the resizing of the control bars when docked. More than one control bar can be docked together, each of them being able to be resized individually using splitters created by the docking system. CAdUiDockControlBar also comes with a gripper bar and a close button when docked. Control bars' state can be switched from docked to undocked or vice versa, by double-clicking on the gripper when docked, or the title bar when undocked, or by dragging them with the mouse. The docking system handles the persistency of the control bars, preserving their position and state across sessions. Finally, CAdUiDockControlBar provides a default context menu to control the bar behavior, with a possibility for the developer to customize this menu.

CAcUiDockControlBar Class

The `CAcUiDockControlBar` class adds to the `CAdUiDockControlBar` class a behavior common to AutoCAD dockable tools: when the user moves the mouse cursor out of the control bar region, the focus is automatically given back to AutoCAD.

AdUi and AcUi Edit Controls

The following classes provide specialized editing controls, including support for specific types of data.

CAdUiEdit Class

`CAdUiEdit` is derived from the `CEdit` class to provide edit box controls. This class provides support for tip windows for truncated text items (TextTips). This class takes bit flags to add desired validation behavior, based on the following types of input: Numeric, String, Angular, and Symbol names. Generally you should use one of the classes derived from the AutoCAD-specific class `CAcUiComboBox`, which adds a specific data type validation and persistency to the control. These are `CAcUiStringEdit`, `CAcUiSymbolEdit`, `CAcUiNumericEdit`, and `CAcUiAngleEdit`.

CAcUiEdit Class

`CAcUiEdit` provides an AutoCAD-specific derivation of `CAdUiEdit`.

CAcUiAngleEdit Class

`CAcUiAngleEdit` is derived from `CAcUiEdit` and provides a specialized constructor to ensure that the `AC_ES_ANGLE` style bit is always set in the style mask. Objects of this class are intended for use in editing angular/rotational data specific to AutoCAD settings.

CAcUiNumericEdit Class

`CAcUiNumericEdit` is derived from `CAcUiEdit` and provides a specialized constructor to ensure that the `AC_ES_NUMERIC` style bit is always set in the style mask. Objects of this class are intended for use in editing numeric data (such as distance) specific to AutoCAD settings.

CAcUiStringEdit Class

`CAcUiStringEdit` is derived from `CAcUiEdit` and provides a specialized constructor to ensure that the `AC_ES_STRING` style bit is always set in the style mask. Any input is acceptable.

CAcUiSymbolEdit Class

`CAcUiSymbolEdit` is derived from `CAcUiEdit` and provides a specialized constructor to ensure that the `AC_ES_SYMBOL` style bit is always set in the style mask. Objects of this class are intended for use in editing valid AutoCAD symbol names.

CAdUiListBox Class

`CAdUiListBox` specializes the MFC `CListBox` to provide a control that supports AdUi messaging. The class can be used anywhere a `CListBox` can be used. Since it provides the additional container-side support for AdUi registered messages, it is convenient to use `CAdUiBaseDialog` (or a derived class) with the `CAdUiListBox` (or a derived class) controls.

`CAdUiListBox` provides features that allow the class to be used to subclass a list box included in a combo box. When used in concert with a `CAdUiComboBox`, the list box is able to track the combo box and, in the case of an owner-draw control, either delegate drawing to the combo box or provide its own drawing routines.

CAdUiListCtrl Class

`CAdUiListCtrl` is derived from `CListCtrl` class to provide list controls. This class provides support for tip windows for truncated text items (TextTips). TextTips will appear for truncated header items for list controls in a report view, and for individual truncated text items in columns in the body of a list control. Owner-drawn controls are supported.

CAdUiHeaderCtrl

`CAdUiHeaderCtrl` specializes `CHeaderCtrl`. Most often, `CAdUiHeaderCtrl` represents the subclassed header contained in a list control (`CAdUiListCtrl`). You do not need to subclass the header control to get TextTip support for column headers in a list control (provided automatically in `CAdUiListCtrl`).

AdUi and AcUi Combo Box Controls

The following classes provide support for combo box controls.

CAdUiComboBox Class

`CAdUiComboBox` is derived from the `CComboBox` class to provide combo box controls. This class provides support for tip windows for truncated text items (TextTips), and data validation in the edit control. This class takes bit flags to add desired validation behavior, based on the following types of input: numeric, string, angular, and symbol names. Generally, you should use one of the classes derived from the AutoCAD-specific class `CAcUiComboBox`, which

adds a specific data type validation and persistency to the control. These are `CAcUiStringComboBox`, `CAcUiSymbolComboBox`, `CAcUiNumericComboBox`, and `CAcUiAngleComboBox`. Support for owner-drawn controls is also built in.

CAcUiAngleComboBox Class

The `CAcUiAngleComboBox` constructor automatically creates a `CAcUiAngleEdit` to subclass the control's edit box. This allows for validation of angles specific to AutoCAD settings.

CAcUiNumericComboBox Class

The `CAcUiAngleComboBox` constructor automatically creates a `CAcUiNumericEdit` to subclass the control's edit box. This allows for validation of numbers specific to AutoCAD settings.

CAcUiStringComboBox Class

The `CAcUiStringComboBox` constructor automatically creates a `CAcUiStringEdit` to subclass the control's edit box. Any input is acceptable.

CAcUiSymbolComboBox Class

The `CAcUiSymbolComboBox` constructor automatically creates a `CAcUiSymbolEdit` to subclass the control's edit box. Valid AutoCAD symbol names are acceptable input.

AcUi MRU Combo Boxes

AcUi extends combo box support to manage an MRU (most recently used) list automatically within the control. The basic functionality is provided by the class `CAcUiMRUComboBox` (derived from `CAcUiComboBox`). A companion class, `CAcUiMRUListBox`, provides DrawTip support for the combo box's `ComboLBox`. This is necessary due to the MRU combo box implementation as an owner-draw control.

Five specialized MRU combo box classes are also provided: `CAcUiArrowHeadComboBox`, `CAcUiColorComboBox`, `CAcUiLineWeightComboBox`, `CAcUiPlotStyleTablesComboBox`, and `CAcUiPlotStyleNamesComboBox`. These provide standard user interfaces for managing dimensioning arrowheads, color and lineweight selections, and plot style table and plot style names selection.

CAcUiMRUComboBox Class

`CAcUiMRUComboBox` inherits `CAcUiComboBox` and serves as the base class for owner-draw combo boxes that implement an MRU list. Each item in the list can contain a small image followed by some text. Each item also tracks a

unique value, referred to as cargo, and maintained as standard Windows® ITEMDATA within the control. The class features built-in support for up to two generic, optional items, referred to as Option1 and Option2. These usually correspond to "ByLayer" and "ByBlock" and often have special significance. Two other items, Other1 and Other2, may also be enabled and appear only when the list is dropped down. Selecting either of these items triggers a special event within the control.

CAcUiArrowHeadComboBox Class

`CAcUiArrowHeadComboBox` specializes `CAcUiMRUComboBox` for dimensioning arrowhead selection. The control displays bitmaps representing the standard AutoCAD dimensioning arrowhead styles, which are always present in the list. By default no optional or additional items are present or added. The cargo associated with each item is the AutoCAD index for the associated stock arrowhead. When MRU items are added to the list, they are automatically assigned a unique cargo value (which will be greater than the AutoCAD index for a user-defined arrowhead style).

CAcUiColorComboBox Class

`CAcUiColorComboBox` specializes `CAcUiMRUComboBox` for color selection. The control displays color swatches representing selections from AutoCAD's palette. The stock items always present in the control reflect color numbers 1 through 7. Both optional items are used; Option1 displays "ByLayer" and Option2 displays "ByBlock". MRU items display "Color nnn," where nnn is the associated color number. The cargo associated with each item indicates an AutoCAD color number (such as 1 to 255), "ByBlock" relates to 0, and "ByLayer" corresponds to 256. The Other1 item is enabled and triggers the AutoCAD Color Selection dialog. If Other2 is enabled it displays as "Windows..." and by default triggers the Windows Color Selection Common dialog. If the user selects an item from either of these dialogs the selection appears in the MRU list and becomes the current item in the control.

CAcUiLineWeightComboBox Class

`CAcUiLineWeightComboBox` specializes `CAcUiMRUComboBox` for lineweight selection. The control displays a small preview of the lineweights AutoCAD supports, ranging from 0.05mm to 2.11mm, and includes "None" and optionally "Default". Both metric and imperial values are displayed, depending on the setting of the LWUNITS system variable. Both optional items are used; Option1 displays "ByLayer" and Option2 displays "ByBlock". Each item maintains cargo that corresponds to the item's `AcDb::kLnWtxxx` value.

CAcUiPlotStyleTablesComboBox Class

`CAcUiPlotStyleTablesComboBox` specializes `CAcUiMRUComboBox` for plot style table selection. The control displays plot style table names according to the current plot style mode (color-dependent mode or named plot styles). The MRU functionality of the combo box is not used. A bitmap indicating an embedded translation table is displayed in named plot style mode for those tables that have an embedded translation table.

CAcUiPlotStyleNamesComboBox Class

`CAcUiPlotStyleNamesComboBox` specializes `CAcUiMRUComboBox` for plot style name selection. The MRU functionality of the combo is not used, and "ByLayer", "ByBlock", and "Other..." items can be conditionally displayed. If present, the "Other..." item can trigger either the Assign Plot Style dialog or the Set Current Plot Style dialog.

CAcUiMRUListBox Class

`CAcUiMRUListBox` derives from `CAcUiListBox`. It is used by `CAcUiMRUComboBox` to subclass the control's list box (`ComboLBox`) and provide DrawTip support. Advanced applications that use specialized MRU combo boxes may need to derive special MRU list boxes to display DrawTips correctly.

AdUi Button Classes

These controls are usable in applications other than AutoCAD.

CAdUiOwnerDrawButton Class

This class provides a basic owner-draw button. The class can be used anywhere a `CButton` can be used. When used in an AdUi-derived dialog (or a class that supports AdUi messaging) `CAdUiOwnerDrawButton` automatically provides for the display of an AdUi tip window. The class also supports drag and drop, Static and Tool Display, and PointedAt effects. In Tool Display mode, the button appears flat and pops up when pointed at (such as when the mouse moves over the button). Clicking the button makes it push down. In Static Display mode, the button appears flat and behaves more like a static control than a push button. The combination of enabling drag and drop and Static Display is appropriate for creating sites that receive files via drag and drop.

CAdUiBitmapButton Class

This class specializes `CAdUiOwnerDrawButton` to provide a button that displays a bitmap (the image is drawn transparently in the button). By default, objects of this class automatically resize to fit the associated bitmap image.

Unlike MFC's `CBitmapButton`, only one bitmap is needed to define all of the button states (MFC's class requires four bitmaps).

CAdUiBitmapStatic Class

`CAdUiBitmapStatic` specializes `CAdUiBitmapButton` to provide a button that enables Static Display by default. These controls act more like statics than pushbuttons.

CAdUiDropSite Class

`CAdUiDropSite` specializes `CAdUiBitmapStatic` to provide a button that enables drag and drop as well as Static Display. These controls can receive files via drag and drop.

CAdUiToolButton Class

`CAdUiToolButton` specializes `CAdUiBitmapButton` to provide a button that enables Tool Display by default. These controls appear more like toolbar buttons than regular pushbuttons.

AcUi Button Classes

These controls build upon the AdUi classes and are usable only with AutoCAD.

CAcUiPickButton Class

`CAcUiPickButton` specializes `CAcUiBitmapButton`, which is a wrapper for the class `CAdUiBitmapButton`. `CAcUiPickButton` provides a button that displays a standard pick button bitmap.

CAcUiSelectButton Class

`CAcUiSelectButton` specializes `CAcUiPickButton`. It provides a button that displays a standard selection button bitmap.

Dialog Data Persistency

`CAcUiDialog` and the `CAcUiTab` classes automatically inherit persistency. Persistency, as defined by the dialogs and controls in *AcUi15.dll*, means that storage for any and all user modal dialogs in AutoCAD derived from these classes will store data with the current user profile, making it a virtual preference.

Your dialog should have a unique name because it will use a shared area of the user profile registry space. Given that developers usually create their

applications using their registered developer prefix, the following method is recommended:

module-name:dialog-name

For example, if your ObjectARX application is named AsdkSample and you have a dialog titled Coordinates, you would name it AsdkSample:Coordinates.

There are two types of dialog data persistency: out-of-the-box and developer-defined. Out-of-the-box persistency refers to dialog position, size, and list view column sizes. Developer-defined refers to any data that a developer chooses to store in the user profile either during the lifetime or dismissal of the dialog and which may be retrieved across dialog invocations.

Using and Extending the AdUi Tab Dialog System

All tabbed dialogs that use `CAdUiTabMainDialog` and `CAdUiTabChildDialog` can be easily made tab extensible. There is no limit for the number of tabs that can be added to a tab-extensible dialog. If the main dialog is resizable, added tabs can participate in that resizing using the same directives outlined in the documentation on resizable dialogs. All dialogs in AutoCAD use scrolling tabs as opposed to stacked tabs.

It is important for you to set the dirty bit for the extended tab using the `SetDirty()` member function of `CAdUiTabChildDialog` when data needs to be initialized or updated via `DoDataExchange`.

Constructing a Custom Tab Dialog That Is Extensible

Construct your tabbed dialog using `CAcUiTabMainDialog` for the main dialog frame and `CAcUiTabChildDialog` for each tab. In the `OnInitDialog()` or constructor of the `CAcUiTabMainDialog` immediately call `SetDialogName()` with the published name of your extensible dialog. ObjectARX applications will use this name to add tabs to your dialog. After you add your tabs with calls to `AddTab()`, in `OnInitDialog`, call `AddExtendedTabs()`. Remember that your tabbed dialog can have any number of added tabs in it, so do not assume a fixed number of tabs elsewhere in the dialog's code.

For example

```
BOOL CPrefTabFrame::OnInitDialog()
// Dialog initialization for my tabbed dialog frame.
{
    SetDialogName("Preferences");
    CAcUiTabMainDialog::OnInitDialog();
    ...

    // Add my tabs here.
    m_tab.AddTab(0,IDS_FILES_TABNAME,IDD_FILES_TAB,&m_filesTab);
    m_tab.AddTab(1,IDS_PERF_TABNAME,IDD_PERF_TAB,&m_performTab);
    m_tab.AddTab(2,IDS_COMP_TABNAME,IDD_COMP_TAB,&m_compatTab);

    // Add any extended tabs. This call is what makes this
    // dialog tab extensible
    AddExtendedTabs();
}
```

Extending the AutoCAD Built-In Tab Dialogs

Use Class Wizard or some other means to create your tab subclassed from `CDialog`. In the properties for the dialog, change the style of the dialog to "popup" and the border to "resizing". Implement an override for `PostNcDestroy()`. Replace all occurrences of `CDialog` with `CAcUiTabExtension` in all source files for the dialog. In `PostNcDestroy()` for the tab extension delete the tab object that has been allocated (see example below).

In your `AcRx::kInitAppMsg` handler in `acrxEntryPoint()` add a call to `acedRegisterExtendedTab("MYAPPNAME.ARX", "DIALOGNAME")`, where `MYAPPNAME` is the base file name of your application and `DIALOGNAME` is the published name of the extensible tabbed dialog you wish to add to.

Implement an `AcRx::kInitDialogMsg` handler in `acrxEntryPoint()` and add the tab there. The `(void*)appId` argument to `acrxEntryPoint()` is a `CAcUiTabExtensionManager` pointer. Use the member function `GetDialogName()` for the `CAcUiTabExtensionManager` to get the name of the dialog being initialized and, if the application wants to add to this dialog, call the `AddTab()` member function of the `CAcUiTabExtensionManager` to add the tab. One argument to this function is a pointer to a previously allocated `CAcUiTabExtension` object. If the dialog is resizable and you want some of your controls to resize, add that resizing code after the call to `AddTab()`.

For example

```
extern "C" AcRx::AppRetCode acrxEntryPoint(
    AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
        break;
    case AcRx::kInitDialogMsg:
    // A dialog is initializing that we are interested in adding
    // tabs to.
        addMyTabs((CAcUiTabExtensionManager*)pkt);
        break;
    default:
        break;
    }
    return AcRx::kRetOK;
}
void initApp()
{
    InitMFC();
    // Do other initialization tasks here.
    acedRegCmds->addCommand(
        "MYARXAPP",
        "MYARXAPP",
        "MYARXAPP",
        ACRX_CMD_MODAL,
        &MyArxAppCreate);
    // Here is where we register the fact that we want to add
    // a tab to the PREFERENCES dialog.
    acedRegisterExtendedTab("MYARXAPP.ARX", "PREFERENCES");
}
// CMyTab1 is subclassed from CAcUiTabExtension.
static CMyTab1* pTab1;
void addMyTabs(CAcUiTabExtensionManager* pXtabManager)
{
    // Allocate an extended tab if it has not been done already
    // and add it through the CAcUiTabExtensionManager.
    pTab1 = new CMyTab1;
    pXtabManager->AddTab(_hdllInstance, IDD_TAB1,
        "My Tab1", pTab1);
    // If the main dialog is resizable, add your control
    // resizing directives here.
    pTab1->StretchControlXY(IDC_EDIT1, 100, 100);
}
```

```
Then for the CMyTab1 class implementation:
void CMyTab1::PostNcDestroy()
// Override to delete added tab.
{
    delete pTab1;
    pTab1 = NULL;
    CAcUiTabExtension::PostNcDestroy();
}
```

Using AdUi and AcUi with VC++ AppWizard

Now that you have seen an overview for the AdUi and AcUi dialog support, we will present an example of using these systems. The dialog we will create will appear as follows. The source code for this example can be found in the ObjectARX SDK *docsamps\AsdkAcUiSample* directory. This example will, however, describe how to set up your project from the beginning.

Create the ObjectARX MFC Application Skeleton

1. Create a new project in Microsoft Visual C++ using the Application Wizard. Choose the MFC AppWizard (dll) project type. Assign the project a name (for this sample we will use the name AsdkAcUiSample) and directory and click OK. On the next screen, choose MFC Extension DLL, then click Finish. We now have a basic MFC Extension DLL project.
2. We will now add the necessary code to support ObjectARX. Open the *AsdkAcUiSample.cpp* file. Remove the `AFX_EXTENSION_MODULE` call and also the `DllMain` function.
3. Add the following declaration:
   ```
   AC_IMPLEMENT_EXTENSION_MODULE(theArxDLL);
   ```
4. Add the following code to set up the AutoCAD command and `acrxEntryPoint`:
   ```
   void dialogCreate()
   {
       acutPrintf("\nAcUi Dialog Sample");
   }
   ```

The following `addCommand` call uses the module resource instance from the `AC_IMPLEMENT_EXTENSION_MODULE` macro:

```
static void initApp()
{
    theArxDLL.AttachInstance();
    CAcModuleResourceOverride resOverride;
    acedRegCmds->addCommand(
        "ASDK_ACUI_SAMPLE",
        "ASDKACUISAMPLE",
        "ACUISAMPLE",
        ACRX_CMD_MODAL,
        dialogCreate,
        NULL,
        -1,
        theArxDLL.ModuleResourceInstance());
}
```

The following `unloadApp()` function is called when the application unloads. At this time it is important to detach the resource instance:

```
static void unloadApp()
{
    // Do other cleanup tasks here
    acedRegCmds->removeGroup("ASDK_ACUI_SAMPLE");

    theArxDLL.DetachInstance();

}
// Entry point
//
extern "C" AcRx::AppRetCode acrxEntryPoint(
    AcRx::AppMsgCode msg, void* appId)
{
    switch( msg )
    {
        case AcRx::kInitAppMsg:
            acrxDynamicLinker->unlockApplication(appId);
            acrxDynamicLinker->registerAppMDIAware(appId);
            initApp();
            break;
        case AcRx::kUnloadAppMsg:
            unloadApp();
            break;
        case AcRx::kInitDialogMsg:
            break;
            default:
            break;
    }
    return AcRx::kRetOK;
}
```

Create an *AsdkAcUiSample.h* header file and add the following lines to the file:

```
#include "resource.h" // main symbols
#define PI  3.14159265359

// Forward declaration for the entry point function of
// our application

void testCreate();
```

Then add the following include files to *AsdkAcUiSample.cpp*:

```
#include "AsdkAcUiSample.h"
#include "AcExtensionModule.h"
```

You will also need to add the ObjectARX libraries to the project file, change the *.dll* extension to *.arx*, and modify the *.def* file with the proper exports. Then you should be able to compile and load the application.

Create the MFC Dialog Using App Studio

1. In Visual C++ App Studio add a dialog resource.
2. Create the following dialog box using the App Studio controls:

3. Make sure the resource IDs match this diagram or the remaining code will not work.

Create the Classes and Controls

1. Using ClassWizard, create the dialog class. If you start ClassWizard from the dialog creation screen it will prompt you to create a new class. Click OK for a new class and then give the dialog a name. For this example use `AsdkAcUiDialogSample`.
2. Switch to the Member Variable tab.
3. For the `IDC_BUTTON_ANGLE` and `IDC_BUTTON_POINT` resources add `CButton` controls called `m_ctrlAngleButton` and `m_ctrlPickButton`, respectively.
4. For the `IDC_EDIT_ANGLE`, `IDC_EDIT_XPT`, `IDC_EDIT_YPT`, and `IDC_EDIT_ZPT` resources add `CEdit` controls called `m_ctrlAngleEdit`, `m_ctrlXPtEdit`, `m_ctrlYPtEdit`, and `m_ctrlZPtEdit`, respectively.
5. For the `IDC_LIST_BLOCKS` resource add a `CListBox` control called `m_ctrlBlockList`.
6. For the `IDC_COMBO_REGAPPS` resource add a `CComboBox` control called `m_ctrlRegAppComboBox`. At this point your member variables dialog should appear like

7. Now open the *AsdkAcUiDialogSample.h* header file and change the derivation of the new dialog class. It should be derived from `CAcUiDialog`:

```
class AsdkAcUiDialogSample : public CAcUiDialog
```

8 Now we will change the types to use the AcUi controls. Start by opening the *AsdkAcUiDialogSample.h* file. Change the control list to be the following:

```
CAcUiSymbolComboBox m_ctrlRegAppComboBox;
CAcUiListBox        m_ctrlBlockListBox;
CAcUiPickButton     m_ctrlPickButton;
CAcUiPickButton     m_ctrlAngleButton;
CAcUiAngleEdit      m_ctrlAngleEdit;
CAcUiNumericEdit    m_ctrlXPtEdit;
CAcUiNumericEdit    m_ctrlYPtEdit;
CAcUiNumericEdit    m_ctrlZPtEdit;
```

9 Also add a couple of member variables to track the point and angle values and some helper functions. These should be added to the public section of the class:

```
AcGePoint3d m_ptValue;
double m_dAngle;
void DisplayPoint();
bool ValidatePoint();
void DisplayAngle();
bool ValidateAngle();
void DisplayBlocks();
void DisplayRegApps();
```

Create the Handlers for the Dialog

1 Go back into ClassWizard and select the Message Maps tab.

2 Highlight the AsdkAcUiDialogSample object ID and add a function for `WM_INITDIALOG`. Then choose edit code to take you into the *AsdkAcUiDialogSample.cpp* source file.

3 Change the parent `OnInitDialog` to be `CAcUiDialog`:

```
CAcUiDialog::OnInitDialog();
```

4 Change the constructor to also initialize `CAcUiDialog`:

```
AsdkAcUiDialogSample::AsdkAcUiDialogSample
(CWnd* pParent /*=NULL*/)
: CAcUiDialog(AsdkAcUiDialogSample::IDD, pParent)
```

The next step is to add message handlers for the `IDC_BUTTON_ANGLE`, `IDC_BUTTON_POINT`, `IDC_COMBO_REGAPPS`, `IDC_EDIT_ANGLE`, and `IDC_OK` resources. Using ClassWizard, add handlers mapped as follows:

Message handlers		
Handler Function	**Resource ID**	**Message**
OnButtonAngle	IDC_BUTTON_ANGLE	BN_CLICKED
OnButtonPoint	IDC_BUTTON_POINT	BN_CLICKED
OnOk	IDOK	BN_CLICKED
OnKillfocusComboRegapps	IDC_COMBO_REGAPPS	CBN_KILLFOCUS
OnKillfocusEditAngle	IDC_EDIT_ANGLE	EN_KILLFOCUS
OnKillfocusEditXpt	IDC_EDIT_XPOINT	EN_KILLFOCUS
OnKillfocusEditYpt	IDC_EDIT_YPOINT	EN_KILLFOCUS
OnKillfocusEditZpt	IDC_EDIT_ZPOINT	EN_KILLFOCUS

Add Code to the Handlers

Once you have added the handlers, you are ready to add code to deal with your dialog. This section summarizes what each handler does with a complete listing.

1 First we add a few utility functions to convert, display, and validate the values. Notice we are using the `CAcUiNumeric` and `CAcUiAngleEdit` controls to do this:

```
// Utility functions
void AsdkAcUiDialogSample::DisplayPoint()
{
    m_ctrlXPtEdit.SetWindowText(m_strXPt);
    m_ctrlXPtEdit.Convert();
    m_ctrlYPtEdit.SetWindowText(m_strYPt);
    m_ctrlYPtEdit.Convert();
    m_ctrlZPtEdit.SetWindowText(m_strZPt);
    m_ctrlZPtEdit.Convert();
}

bool AsdkAcUiDialogSample::ValidatePoint()
{
    if (!m_ctrlXPtEdit.Validate())
        return false;
    if (!m_ctrlYPtEdit.Validate())
        return false;
    if (!m_ctrlZPtEdit.Validate())
        return false;
    return true;
}
```

```
void AsdkAcUiDialogSample::DisplayAngle()
{
    m_ctrlAngleEdit.SetWindowText(m_strAngle);
    m_ctrlAngleEdit.Convert();
}

bool AsdkAcUiDialogSample::ValidateAngle()
{
    if (!m_ctrlAngleEdit.Validate())
        return false;

    return true;
}
```

2 Now add some utility functions to iterate over two symbol tables and display the names in the two different list boxes:

```
void AsdkAcUiDialogSample::DisplayBlocks()
{
    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    // Iterate through the block table and display
    // the names in the list box.
    //
    char *pName;
    AcDbBlockTableIterator *pBTItr;
    if (pBlockTable->newIterator(pBTItr) == Acad::eOk) {
        while (!pBTItr->done()) {
            AcDbBlockTableRecord *pRecord;
            if (pBTItr->getRecord(pRecord, AcDb::kForRead)
                == Acad::eOk) {
                pRecord->getName(pName);
                m_ctrlBlockListBox.InsertString(-1, pName);
                pRecord->close();
            }
            pBTItr->step();
        }
    }
    pBlockTable->close();
}

void AsdkAcUiDialogSample::DisplayRegApps()
{
    AcDbRegAppTable *pRegAppTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pRegAppTable, AcDb::kForRead);

    // Iterate through the reg app table and display the
    // names in the list box.
    //
```

```
    char *pName;
    AcDbRegAppTableIterator *pItr;
    if (pRegAppTable->newIterator(pItr) == Acad::eOk) {
        while (!pItr->done()) {
            AcDbRegAppTableRecord *pRecord;
            if (pItr->getRecord(pRecord, AcDb::kForRead)
                == Acad::eOk) {
                pRecord->getName(pName);
                m_ctrlRegAppComboBox.InsertString(-1, pName);
                pRecord->close();
            }
            pItr->step();
        }
    }
    pRegAppTable->close();
}
```

3 Add the declarations for the functions and variables to the class definition in the header file:

```
void DisplayPoint();
bool ValidatePoint();
void DisplayAngle();
bool ValidateAngle();
void DisplayBlocks();
void DisplayRegApps();
CString m_strAngle;
CString m_strXPt;
CString m_strYPt;
CString m_strZPt;
```

4 Next are the button handlers for picking a point and angle using the AutoCAD editor. Notice how the `BeginEditorCommand()`, `CompleteEditorCommand()`, and `CancelEditorCommand()` functions are used to hide the dialog, allow the call to `acedGetPoint` and `acedGetAngle`, and finally either cancel or redisplay the dialog based on how the user picked:

```
// AsdkAcUiDialogSample message handlers
void AsdkAcUiDialogSample::OnButtonPoint()
{
    // Hide the dialog and give control to the editor
    //
    BeginEditorCommand();
    ads_point pt;

    // Get a point
    //
```

```
    if (acedGetPoint(NULL, "\nPick a point: ", pt) == RTNORM) {

        // If the point is good, continue
        //
        CompleteEditorCommand();
        m_strXPt.Format("%g", pt[X]);
        m_strYPt.Format("%g", pt[Y]);
        m_strZPt.Format("%g", pt[Z]);
        DisplayPoint();
    } else {

        // otherwise cancel the command (including the dialog)
        CancelEditorCommand();
    }
}

void AsdkAcUiDialogSample::OnButtonAngle()
{
    // Hide the dialog and give control to the editor
    //
    BeginEditorCommand();

    // Set up the default point for picking an angle
    // based on the m_strXPt, m_strYPt, and m_strZPt values
    //
    ads_point pt;
    acdbDisToF(m_strXPt, -1, &pt[X]);
    acdbDisToF(m_strYPt, -1, &pt[Y]);
    acdbDisToF(m_strZPt, -1, &pt[Z]);

    double angle;

    // Get a point from the user
    //
    if (acedGetAngle(pt, "\nPick an angle: ", &angle) == RTNORM) {
        // If we got an angle, go back to the dialog
        //
        CompleteEditorCommand();

        // Convert the acquired radian value to degrees since the
        // AcUi control can convert that to the other formats.
        //
        m_strAngle.Format("%g", angle*(180.0/PI));
        DisplayAngle();
    } else {
    // otherwise cancel the command (including the dialog)
    //
        CancelEditorCommand();
    }
}
```

5 Now the edit box handlers are implemented. Basically we just want to convert the values to the current Units settings:

```
void AsdkAcUiDialogSample::OnKillfocusEditAngle()
{
    // Get and update text the user typed in.
    //
    m_ctrlAngleEdit.Convert();
    m_ctrlAngleEdit.GetWindowText(m_strAngle);
}

void AsdkAcUiDialogSample::OnKillfocusEditXpt()
{
    // Get and update text the user typed in.
    //
    m_ctrlXPtEdit.Convert();
    m_ctrlXPtEdit.GetWindowText(m_strXPt);
}

void AsdkAcUiDialogSample::OnKillfocusEditYpt()
{
    // Get and update text the user typed in.
    //
    m_ctrlYPtEdit.Convert();
    m_ctrlYPtEdit.GetWindowText(m_strYPt);
}

void AsdkAcUiDialogSample::OnKillfocusEditZpt()
{
    // Get and update text the user typed in.
    //
    m_ctrlZPtEdit.Convert();
    m_ctrlZPtEdit.GetWindowText(m_strZPt);
}
```

6 The combo box handler allows the user to type in a string and then register this as an application name. This doesn't really make sense for an application, but it shows the use of a combo box:

```
void AsdkAcUiDialogSample::OnKillfocusComboRegapps()
{
    CString strFromEdit;
    m_ctrlRegAppComboBox.GetWindowText(strFromEdit);
    if (m_ctrlRegAppComboBox.FindString(-1, strFromEdit) == CB_ERR)
        if (acdbRegApp(strFromEdit) == RTNORM)
            m_ctrlRegAppComboBox.AddString(strFromEdit);
}
```

7 To do some data validation, we handle this in the `OnOk()` handler. This, of course, can be done at any time. Also notice that the `OnOk()` handler is storing the data into the user profile (registry) using the `SetDialogData()` function:

```
void AsdkAcUiDialogSample::OnOK()
{
    if (!ValidatePoint()) {
        AfxMessageBox("Sorry, Point out of desired range.");
        m_ctrlXPtEdit.SetFocus();
        return;
    }

    if (!ValidateAngle()) {
        AfxMessageBox("Sorry, Angle out of desired range.”);
        m_ctrlAngleEdit.SetFocus();
        return;
    }

    // Store the data into the registry
    //
    SetDialogData("ANGLE", m_strAngle);
    SetDialogData("POINTX", m_strXPt);
    SetDialogData("POINTY", m_strYPt);
    SetDialogData("POINTZ", m_strZPt);
    CAcUiDialog::OnOK();
}
```

8 Finally, the `OnInitDialog()` function takes care of all the initialization, including the resizing and data persistency requirements:

```
BOOL AsdkAcUiDialogSample::OnInitDialog()
{
    // Set the dialog name for registry lookup and storage
    //
    SetDialogName("AsdkAcUiSample:AsdkAcUiDialog");
    CAcUiDialog::OnInitDialog();

    DLGCTLINFOdlgSizeInfo[]= {
        { IDC_STATIC_GROUP1, ELASTICX, 20 },
        { IDC_STATIC_GROUP1, ELASTICY, 100 },
        { IDC_EDIT_XPT,ELASTICX, 20 },
        { IDC_EDIT_YPT,ELASTICX, 20 },
        { IDC_EDIT_ZPT,ELASTICX, 20 },
        { IDC_EDIT_ANGLE, ELASTICX, 20 },
        { IDC_STATIC_GROUP2, MOVEX, 20 },
        { IDC_STATIC_GROUP2, ELASTICY, 100 },
        { IDC_STATIC_GROUP2, ELASTICX, 80 },
        { IDC_LIST_BLOCKS, MOVEX, 20 },
        { IDC_LIST_BLOCKS, ELASTICY, 100 },
        { IDC_STATIC_TEXT2,MOVEX, 20 },
        { IDC_STATIC_TEXT2,MOVEY, 100 },
        { IDC_LIST_BLOCKS, ELASTICX, 80 },
        { IDC_STATIC_TEXT2,ELASTICX, 80 },
```

```
        { IDC_STATIC_GROUP3, MOVEY, 100 },
        { IDC_STATIC_GROUP3, ELASTICX, 20 },
        { IDC_COMBO_REGAPPS, MOVEY, 100 },
        { IDC_COMBO_REGAPPS, ELASTICX, 20 },
        { IDC_STATIC_TEXT3,MOVEY, 100 },
        { IDC_STATIC_TEXT3,ELASTICX, 20 },
        { IDOK,MOVEX, 100 },
        { IDCANCEL, MOVEX, 100 },
    };

    const DWORD numberofentries =
        sizeof dlgSizeInfo / sizeof DLGCTLINFO;
    SetControlProperty(dlgSizeInfo, numberofentries);

    // Must be within a 100-unit cube centered about 0,0,0.
    //
    m_ctrlXPtEdit.SetRange(-50.0, 50.0);
    m_ctrlYPtEdit.SetRange(-50.0, 50.0);
    m_ctrlZPtEdit.SetRange(-50.0, 50.0);

    // Must be between 0 and 90 degrees.
    //
    m_ctrlAngleEdit.SetRange(0.0, 90.0 /*(PI/2.0)*/);

    // Assign a title for the dialog.
    //
    SetWindowText("AcUiDialog Sample");

    // Load the default bitmaps.
    //
    m_ctrlPickButton.AutoLoad();
    m_ctrlAngleButton.AutoLoad();

    // Get and display the preserved data from the registry.
    //
    if (!GetDialogData("ANGLE", m_strAngle))
        m_strAngle = "0.0";
    if (!GetDialogData("POINTX", m_strXPt))
        m_strXPt = "0.0";
    if (!GetDialogData("POINTY", m_strYPt))
        m_strYPt = "0.0";
    if (!GetDialogData("POINTZ", m_strZPt))
        m_strZPt = "0.0";

    DisplayPoint();
    DisplayAngle();
    DisplayBlocks();
    DisplayRegApps();

    return TRUE; // return TRUE unless you set the focus to a control
}
```

Selection Set, Entity, and Symbol Table Functions

In This Chapter

9

- Selection Set and Entity Names
- Handling Selection Sets
- Entity Name and Data Functions
- Symbol Table Access

The global functions described in this chapter handle selection sets, drawing entities, and symbol tables. See the *AutoCAD Customization Guide* for background information on these topics.

Selection Set and Entity Names

Most of the ObjectARX functions that handle selection sets and entities identify a set or entity by its name, which is a pair of longs assigned and maintained by AutoCAD. In ObjectARX, names of selection sets and entities have the corresponding type `ads_name`.

Before it can manipulate a selection set or an entity, an ObjectARX application must obtain the current name of the set or entity by calling one of the library functions that returns a selection set or entity name.

NOTE Selection set and entity names are volatile; they apply only while you are working on a drawing with AutoCAD, and they are lost when exiting from AutoCAD or switching to another drawing.

For selection sets, which also apply only to the current session, the volatility of names poses no problem, but for entities, which are saved in the drawing database, it does. An application that must refer at different times to the same entities in the same drawing (or drawings), can use entity handles, described in "Entity Handles and Their Uses" on page 216.

Handling Selection Sets

The ObjectARX functions that handle selection sets are similar to those in AutoLISP. The `acedSSGet()` function provides the most general means of creating a selection set. It creates a selection set in one of three ways:

- Prompting the user to select objects.
- Explicitly specifying the entities to select by using the PICKFIRST set or the Crossing, Crossing Polygon, Fence, Last, Previous, Window, or Window Polygon options (as in interactive AutoCAD use), or by specifying a single point or a fence of points.
- Filtering the current drawing database by specifying a list of attributes and conditions that the selected entities must match. You can use filters with any of the previous options.

```
int
acedSSGet (
    const char *str,
    const void *pt1,
    const void *pt2,
    const struct resbuf *entmask,
    ads_name ss);
```

The first argument to `acedSSGet()` is a string that describes which selection options to use, as summarized in the following table.

Selection options for acedSSGet

Selection Code	Description
NULL	Single-point selection (if pt1 is specified) or user selection (if pt1 is also NULL)
#	Nongeometric (all, last, previous)
:$	Prompts supplied
.	User pick
:?	Other callbacks
A	All
B	Box
C	Crossing
CP	Crossing Polygon
:D	Duplicates OK
:E	Everything in aperture
F	Fence
G	Groups
I	Implied
:K	Keyword callbacks
L	Last
M	Multiple
P	Previous
:S	Force single object selection only
W	Window
WP	Window Polygon
X	Extended search (search whole database)

The next two arguments specify point values for the relevant options. (They should be NULL if they don't apply.) If the fourth argument, entmask, is not NULL, it points to the list of entity field values used in filtering. The fifth argument, ss, identifies the selection set's name.

The following code shows representative calls to acedSSGet(). As the acutBuildList() call illustrates, for the polygon options "CP" and "WP" (but not for "F"), acedSSGet() automatically closes the list of points. You don't need to build a list that specifies a final point identical to the first.

```
ads_point pt1, pt2, pt3, pt4;
struct resbuf *pointlist;
ads_name ssname;
pt1[X] = pt1[Y] = pt1[Z] = 0.0;
pt2[X] = pt2[Y] = 5.0; pt2[Z] = 0.0;

// Get the current PICKFIRST set, if there is one;
// otherwise, ask the user for a general entity selection.
acedSSGet(NULL, NULL, NULL, NULL, ssname);

// Get the current PICKFIRST set, if there is one.
acedSSGet("I", NULL, NULL, NULL, ssname);

// Selects the most recently selected objects.
acedSSGet("P", NULL, NULL, NULL, ssname);

// Selects the last entity added to the database.
acedSSGet("L", NULL, NULL, NULL, ssname);

// Selects entity passing through point (5,5).
acedSSGet(NULL, pt2, NULL, NULL, ssname);

// Selects entities inside the window from (0,0) to (5,5).
acedSSGet("W", pt1, pt2, NULL, ssname);

// Selects entities enclosed by the specified polygon.
pt3[X] = 10.0; pt3[Y] = 5.0; pt3[Z] = 0.0;
pt4[X] = 5.0; pt4[Y] = pt4[Z] = 0.0;
pointlist = acutBuildList(RTPOINT, pt1, RTPOINT, pt2,
            RTPOINT, pt3, RTPOINT, pt4, 0);
acedSSGet("WP", pointlist, NULL, NULL, ssname);

// Selects entities crossing the box from (0,0) to (5,5).
acedSSGet("C", pt1, pt2, NULL, ssname);
```

```
// Selects entities crossing the specified polygon.
acedSSGet("CP", pointlist, NULL, NULL, ssname);
acutRelRb(pointlist);

// Selects the entities crossed by the specified fence.
pt4[Y] = 15.0; pt4[Z] = 0.0;
pointlist = acutBuildList(RTPOINT, pt1, RTPOINT, pt2,
            RTPOINT, pt3, RTPOINT, pt4, 0);
acedSSGet("F", pointlist, NULL, NULL, ssname);
acutRelRb(pointlist);
```

The complement of `acedSSGet()` is `acedSSFree()`, which releases a selection set once the application has finished using it. The selection set is specified by name. The following code fragment uses the `ads_name` declaration from the previous example.

```
acedSSFree(ssname);
```

NOTE AutoCAD cannot have more than 128 selection sets open at once. This limit includes the selection sets open in all concurrently running ObjectARX and AutoLISP applications. The limit may be different on your system. If the limit is reached, AutoCAD refuses to create more selection sets. Simultaneously managing a large number of selection sets is not recommended. Instead, keep a reasonable number of sets open at any given time, and call `acedSSFree()` to free unused selection sets as soon as possible. Unlike AutoLISP, the ObjectARX environment has no automatic garbage collection to free selection sets after they have been used. An application should always free its open selection sets when it receives a `kUnloadDwgMsg`, `kEndMsg`, or `kQuitMsg` message.

Selection Set Filter Lists

When the `entmask` argument specifies a list of entity field values, `acedSSGet()` scans the selected entities and creates a selection set containing the names of all main entities that match the specified criteria. For example, using this mechanism, you can obtain a selection set that includes all entities of a given type, on a given layer, or of a given color.

You can use a filter in conjunction with any of the selection options. The "x" option says to create the selection set using only filtering; as in previous AutoCAD versions, if you use the "x" option, `acedSSGet()` scans the entire drawing database.

NOTE If only filtering is specified ("x") but the `entmask` argument is `NULL`, `acedSSGet()` selects all entities in the database.

The `entmask` argument must be a result buffer list. Each buffer specifies a property to check and a value that constitutes a match; the buffer's `restype` field is a DXF group code that indicates the kind of property to look for, and its `resval` field specifies the value to match.

The following are some examples.

```
struct resbuf eb1, eb2, eb3;
char sbuf1[10], sbuf2[10]; // Buffers to hold strings
ads_name ssname1, ssname2;

eb1.restype = 0;// Entity name
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1;
eb1.rbnext = NULL; // No other properties

// Retrieve all circles.
acedSSGet("X", NULL, NULL, &eb1, ssname1);

eb2.restype = 8; // Layer name
strcpy(sbuf2, "FLOOR3");
eb2.resval.rstring = sbuf2;
eb2.rbnext = NULL; // No other properties

// Retrieve all entities on layer FLOOR3.
acedSSGet("X", NULL, NULL, &eb2, ssname2);
```

NOTE The `resval` specified in each buffer must be of the appropriate type. For example, name types are strings (`resval.rstring`); elevation and thickness are double-precision floating-point values (`resval.rreal`); color, attributes-follow, and flag values are short integers (`resval.rint`); extrusion vectors are three-dimensional points (`resval.rpoint`); and so forth.

If `entmask` specifies more than one property, an entity is included in the selection set only if it matches *all* specified conditions, as shown in the following example:

```
eb3.restype = 62; // Entity color
eb3.resval.rint = 1; // Request red entities.
eb3.rbnext = NULL; // Last property in list

eb1.rbnext = &eb2; // Add the two properties
eb2.rbnext = &eb3; // to form a list.

// Retrieve all red circles on layer FLOOR3.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

An entity is tested against all fields specified in the filtering list unless the list contains relational or conditional operators, as described in "Relational Tests" on page 207 and "Conditional Filtering" on page 208.

The `acedSSGet()` function returns `RTERROR` if no entities in the database match the specified filtering criteria.

The previous `acedSSGet()` examples use the "x" option, which scans the entire drawing database. If filter lists are used in conjunction with the other options (user selection, a window, and so forth), the filter is applied only to the entities initially selected.

The following is an example of the filtering of user-selected entities.

```
eb1.restype = 0; // Entity type group
strcpy(sbuf1, "TEXT");
eb1.resval.rstring = sbuf1; // Entity type is text.
eb1.rbnext = NULL;

// Ask the user to generally select entities, but include
// only text entities in the selection set returned.
acedSSGet(NULL, NULL, NULL, &eb1, ssname1);
```

The next example demonstrates the filtering of the previous selection set.

```
eb1.restype = 0; // Entity type group
strcpy(sbuf1, "LINE");
eb1.resval.rstring = sbuf1; // Entity type is line.
eb1.rbnext = NULL;

// Select all the lines in the previously created selection set.
acedSSGet("P", NULL, NULL, &eb1, ssname1);
```

The final example shows the filtering of entities within a selection window.

```
eb1.restype = 8; // Layer
strcpy(sbuf1, "FLOOR9");
eb1.resval.rstring = sbuf1; // Layer name
eb1.rbnext = NULL;

// Select all the entities within the window that are also
// on the layer FLOOR9.
acedSSGet("W", pt1, pt2, &eb1, ssname1);
```

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 that do not contain that group code are excluded from the selection sets that `acedSSGet()` returns.

Wild-Card Patterns in Filter Lists

Symbol names specified in filter lists can include wild-card patterns. The wild-card patterns recognized by `acedSSGet()` are the same as those recognized by the function `acutWcMatch()`.

The following sample code retrieves an anonymous block named `*U2`.

```
eb2.restype = 2; // Block name
strcpy(sbuf1, "'*U2"); // Note the reverse quote.
eb2.resval.rstring = sbuf1; // Anonymous block name
eb2.rbnext = NULL;

// Select Block Inserts of the anonymous block *U2.
acedSSGet("X", NULL, NULL, &eb2, ssname1);
```

Filtering for Extended Data

Extended data (xdata) are text strings, numeric values, 3D points, distances, layer names, or other data attached to an object, typically by an external application.

The size of extended data is 16K bytes.

You can retrieve extended data for a particular application by specifying its name in a filter list, using the -3 group code. The `acedSSGet()` function returns entities with extended data registered to the specified name; `acedSSGet()` does not retrieve individual extended data items (with group codes in the range 1000–2000).

The following sample code fragment selects all circles that have extended data registered to the application whose ID is "`APPNAME`".

```
eb1.restype = 0; // Entity type
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1; // Circle
eb1.rbnext = &eb2;

eb2.restype = -3; // Extended data
eb2.rbnext = &eb3;
eb3.restype = 1001;
strcpy(sbuf2, "APPNAME");
eb3.resval.rstring = sbuf2; // APPNAME application
eb3.rbnext = NULL;

// Select circles with XDATA registered to APPNAME.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

If more than one application name is included in the list, `acedSSGet()` includes an entity in the selection set only if it has extended data for all the specified applications. For example, the following code selects circles with extended data registered to "`APP1`" and "`APP2`".

```
eb1.restype = 0; // Entity type
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1; // Circle
eb1.rbnext = &eb2;

eb2.restype = -3; // Extended data
eb2.rbnext = &eb3;
```

```
eb3.restype = 1001;
strcpy(sbuf2, "APP1");
eb2.resval.rstring = sbuf2; // APP1 application
eb2.rbnext = &eb4;

eb4.restype = 1001; // Extended data
strcpy(sbuf3, "APP2");
eb4.resval.rstring = sbuf3; // APP2 application
eb4.rbnext = NULL;

// Select circles with XDATA registered to APP1 & APP2.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

You can specify application names using wild-card strings, so you can search for the data of multiple applications at one time. For example, the following code selects all circles with extended data registered to "`APP1`" or "`APP2`" (or both).

```
eb1.restype = 0; // Entity type
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1; // Circle
eb1.rbnext = &eb2;

eb2.restype = -3; // Extended data
eb2.rbnext = &eb3;
eb3.restype = 1001; // Extended data
strcpy(sbuf2, "APP1,APP2");
eb3.resval.rstring = sbuf2; // Application names
eb3.rbnext = NULL;

// Select circles with XDATA registered to APP1 or APP2.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

The following string finds extended data of the same application.

```
strcpy(sbuf2, "APP[12]");
```

Relational Tests

Unless you specify otherwise, there is an implied "equals" test between the entity and each item in the filter list. For numeric groups (integers, real values, points, and vectors), you can specify other relations by including relational operators in the filter list. Relational operators are passed as a special -4 group, whose value is a string that indicates the test to be applied to the next group in the filter list.

The following sample code selects all circles whose radii are greater than or equal to 2.0:

```
eb3.restype = 40; // Radius
eb3.resval.rreal = 2.0;
eb3.rbnext = NULL;
```

```
eb2.restype = -4; // Filter operator
strcpy(sbuf1, ">=");
eb2.resval.rstring = sbuf1; // Greater than or equals
eb2.rbnext = &eb3;

eb1.restype = 0; // Entity type
strcpy(sbuf2, "CIRCLE");
eb1.resval.rstring = sbuf2; // Circle
eb1.rbnext = &eb2;

// Select circles whose radius is >= 2.0.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

Conditional Filtering

The relational operators just described are binary operators. You can also test groups by creating nested Boolean expressions that use conditional operators. The conditional operators are also specified by -4 groups, but they must be paired.

The following sample code selects all circles in the drawing with a radius of 1.0 and all lines on the layer "ABC".

```
eb1 = acutBuildList(-4, "<or",-4, "<and", RTDXF0,
    "CIRCLE", 40, 1.0, -4, "and>", -4, "<and", RTDXF0,
    "LINE", 8, "ABC", -4, "and>", -4, "or>", 0);

acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

The conditional operators are not case sensitive; you can use lowercase equivalents.

NOTE Conditional expressions that test for extended data using the -3 group can contain only -3 groups. See "Filtering for Extended Data" on page 206.

To select all circles that have extended data registered to either "APP1" or "APP2" but not both, you could use the following code.

```
eb1 = acutBuildList(-4, "<xor", -3, "APP1", -3, "APP2",
    -4, "xor>", 0);

acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

Selection Set Manipulation

You can add entities to a selection set or remove them from it by calling the functions `acedSSAdd()` and `acedSSDel()`, which are similar to the Add and Remove options when AutoCAD interactively prompts the user to select objects or remove objects.

NOTE The `acedSSAdd()` function can also be used to create a new selection set, as shown in the following example. As with `acedSSGet()`, `acedSSAdd()` creates a new selection set only if it returns `RTNORM`.

The following sample code fragment creates a selection set that includes the first and last entities in the current drawing.

```
ads_name fname, lname; // Entity names
ads_name ourset; // Selection set name

// Get the first entity in the drawing.
if (acdbEntNext(NULL, fname) != RTNORM) {
    acdbFail("No entities in drawing\n");
    return BAD;
}

// Create a selection set that contains the first entity.
if (acedSSAdd(fname, NULL, ourset) != RTNORM) {
    acdbFail("Unable to create selection set\n");
    return BAD;
}

// Get the last entity in the drawing.
if (acdbEntLast(lname) != RTNORM) {
    acdbFail("No entities in drawing\n");
    return BAD;
}
// Add the last entity to the same selection set.
if (acedSSAdd(lname, ourset, ourset) != RTNORM) {
    acdbFail("Unable to add entity to selection set\n");
    return BAD;

}
```

The example runs correctly even if there is only one entity in the database (in which case both `acdbEntNext()` and `acdbEntLast()` set their arguments to the same entity name). If `acedSSAdd()` is passed the name of an entity that is already in the selection set, it ignores the request and does not report an error.

As the example also illustrates, the second and third arguments to `acedSSAdd()` can be passed as the same selection set name. That is, if the call is successful, the selection set named by both arguments contains an addi-

tional member after `acedSSAdd()` returns (unless the specified entity was already in the selection set).

The following call removes the entity with which the selection set was created in the previous example.

```
acedSSDel(fname, ourset);
```

If there is more than one entity in the drawing (that is, if `fname` and `lname` are not equal), the selection set `ourset` now contains only `lname`, the last entity in the drawing.

The function `acedSSLength()` returns the number of entities in a selection set, and `acedSSMemb()` tests whether a particular entity is a member of a selection set. Finally, the function `acedSSName()` returns the name of a particular entity in a selection set, using an index into the set (entities in a selection set are numbered from 0).

NOTE Because selection sets can be quite large, the `len` argument returned by `acedSSLength()` must be declared as a `long` integer. The `i` argument used as an index in calls to `acedSSName()` must also be a `long` integer. (In this context, standard C compilers will correctly convert a plain integer.)

The following sample code shows a few calls to `acedSSName()`.

```
ads_name sset, ent1, ent4, lastent;
long ilast;

// Create the selection set (by prompting the user).
acedSSGet(NULL, NULL, NULL, NULL, sset);

// Get the name of first entity in sset.
if (acedSSName(sset, 0L, ent1) != RTNORM)
    return BAD;

// Get the name of the fourth entity in sset.
if (acedSSName(sset, 3L, ent4) != RTNORM) {
    acdbFail("Need to select at least four entities\n");
    return BAD;
}

// Find the index of the last entity in sset.
if (acedSSLength(sset, &ilast) != RTNORM)
    return BAD;

// Get the name of the last entity in sset.
if (acedSSName(sset, ilast-1, lastent) != RTNORM)
    return BAD;
```

Transformation of Selection Sets

The function `acedXformSS()` transforms a selection set by applying a transformation matrix (of type `ads_matrix`) to the entities in the set. This provides an efficient alternative to invoking the ROTATE, SCALE, MIRROR, or MOVE commands with `acedCommand()` (or `acedCmd()`) or to changing values in the database with `acdbEntMod()`. The selection set can be obtained in any of the usual ways. The matrix must do uniform scaling. That is, the elements in the scaling vector S_X S_Y S_Z must all be equal; in matrix notation, M_{00} M_{11} M_{22}.

If the scale vector is not uniform, `acedXformSS()` reports an error.

The following sample code gets a selection set by using a crossing box, and then applies the following matrix to it.

$$\begin{bmatrix} 0.5 & 0.0 & 0.0 & 20.0 \\ 0.0 & 0.5 & 0.0 & 5.0 \\ 0.0 & 0.0 & 0.5 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Applying this matrix scales the entities by one-half (which moves them toward the origin) and translates their location by (20.0,5.0).

```
int rc, i, j;
ads_point pt1, pt2;
ads_matrix matrix;
ads_name ssname;

// Initialize pt1 and pt2 here.

rc = acedSSGet("C", pt1, pt2, NULL, ssname);
if (rc == RTNORM) {
// Initialize to identity.
    ident_init(matrix);
// Initialize scale factors.
    matrix[0][0] = matrix[1][1] = matrix[2][2] = 0.5;
```

```
// Initialize translation vector.
    matrix[0][T] = 20.0;
    matrix[1][T] = 5.0;

    rc = acedXformSS(ssname, matrix);
}
```

When you invoke `acedDragGen()`, you must specify a similar function to let users interactively control the effect of the transformation. The function's declaration must have the following form:

```
int scnf(ads_point pt, ads_matrix mt)
```

It should return `RTNORM` if it modified the matrix, `RTNONE` if it did not, or `RTERROR` if it detects an error.

The `acedDragGen()` function calls the `scnf` function every time the user moves the cursor. The `scnf()` function sets the new value of the matrix `mt`. When `scnf()` returns with a status of `RTNORM`, `acedDragGen()` applies the new matrix to the selection set. If there is no need to modify the matrix (for example, if `scnf()` simply displays transient vectors with `acedGrVecs()`), `scnf()` should return `RTNONE`. In this case, `acedDragGen()` ignores `mt` and doesn't transform the selection set.

In the following example, the function sets the matrix to simply move (translate) the selection set without scaling or rotation.

```
int dragsample(usrpt, matrix)
ads_point usrpt
ads_matrix matrix;
{
    ident_init(matrix); // Initialize to identity.
// Initialize translation vector.
    matrix[0][T] = usrpt[X];
    matrix[1][T] = usrpt[Y];
    matrix[2][T] = usrpt[Z];

    return RTNORM; // Matrix was modified.
}
```

Conversely, the following version of `dragsample()` scales the selection set in the current *XY* plane but doesn't move it.

```
int dragsample(usrpt, matrix)
ads_point usrpt
ads_matrix matrix;
{
    ident_init(matrix); // Initialize to identity.
    matrix[0][0] = userpt[X];
    matrix[1][1] = userpt[Y];

    return RTNORM; // Matrix was modified.
}
```

A call to `acedDragGen()` that employs the transformation function looks like this:

```
int rc;
ads_name ssname;
ads_point return_pt;

// Prompt the user for a general entity selection:
if (acedSSGet(NULL, NULL, NULL, NULL, ssname) == RTNORM)

    rc = acedDragGen(ssname, // The new entities
        "Scale the selected objects by dragging", // Prompt
         0, // Display normal cursor (crosshairs)
         dragsample, // Pointer to the transform function
         return_pt); // Set to the specified location
```

More complex transformations can rotate entities, combine transformations (as in the `acedXformSS()` example), and so forth.

Combining transformation matrices is known as matrix composition. The following function composes two transformation matrices by returning their product in `resmat`.

```
void xformcompose(ads_matrix xf1, ads_matrix xf2,
        ads_matrix resmat)
{
    int i, j, k;
    ads_real sum;

    for (i=0; i<=3; i++) {
        for (j=0; j<=3; j++) {
            sum = 0.0;
            for (k=0; k<3; k++) {
                sum += xf1[i,k] * xf2[k,j];
            }
            resmat[i,j] = sum;
        }
    }
}
```

Entity Name and Data Functions

Entity-handling functions are organized into two categories: functions that retrieve the name of a particular entity and functions that retrieve or modify entity data.

Entity Name Functions

To operate on an entity, an ObjectARX application must obtain its name for use in subsequent calls to the entity data functions or the selection set functions. The functions `acedEntSel()`, `acedNEntSelP()`, and `acedNEntSel()` return not only the entity's name but additional information for the application's use. The `entsel` functions require AutoCAD users (or the application) to select an entity by specifying 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 is on a frozen layer. Like the `acedGetxxx()` functions, you can have `acedEntSel()`, `acedNEntSelP()`, and `acedNEntSel()` return a keyword instead of a point by preceding them with a call to `acedInitGet()`.

If a call to `acedEntSel()`, `acedNEntSelP()`, or `acedNEntSel()` returns RTERROR, and you want to know whether the user specified a point that had no entity or whether the user pressed RETURN, you can inspect the value of the ERRNO system variable. If the user specified an empty point, ERRNO equals 7 (OL_ENTSELPICK). If the user pressed RETURN, ERRNO equals 52 (OL_ENTSELNULL). (You can use the symbolic names if your program includes the header file ol_*errno.h*.)

NOTE You should inspect ERRNO immediately after `acedEntSel()`, `acedNEntSelP()`, or `acedNEntSel()` returns. A subsequent ObjectARX call can change the value of ERRNO.

The `acdbEntNext()` function retrieves entity names sequentially. If its first argument is `NULL`, it returns the name of the first entity in the drawing database; if its first argument is the name of an entity in the current drawing, it returns the name of the succeeding entity.

The following sample code fragment illustrates how `acedSSAdd()` can be used in conjunction with `acdbEntNext()` to create selection sets and to add members to an existing set.

```
ads_name ss, e1, e2;
```

```
// Set e1 to the name of first entity.
if (acdbEntNext(NULL, e1) != RTNORM) {
    acdbFail("No entities in drawing\n");
    return BAD;
}

// Set ss to a null selection set.
acedSSAdd(NULL, NULL, ss);

// Return the selection set ss with entity name e1 added.
if (acedSSAdd(e1, ss, ss) != RTNORM) {
    acdbFail("Unable to add entity to selection set\n");
    return BAD;
}

// Get the entity following e1.
if (acdbEntNext(e1, e2) != RTNORM) {
    acdbFail("Not enough entities in drawing\n");
    return BAD;
}

// Add e2 to selection set ss
if (acedSSAdd(e2, ss, ss) != RTNORM) {
    acdbFail("Unable to add entity to selection set\n");
    return BAD;
}
```

The following sample code fragment uses `acdbEntNext()` to "walk" through the database, one entity at a time.

```
ads_name ent0, ent1;
struct resbuf *entdata;

if (acdbEntNext(NULL, ent0) != RTNORM) {
    acdbFail("Drawing is empty\n");
    return BAD;
}
do {
// Get entity's definition data.
    entdata = acdbEntGet(ent0);
    if (entdata == NULL) {
        acdbFail("Failed to get entity\n");
        return BAD;
    }
        .
        . // Process new entity.
        .
    if (acedUsrBrk() == TRUE) {
        acdbFail("User break\n");
        return BAD;
    }
    acutRelRb(entdata); // Release the list.
    ads_name_set(ent0, ent1); // Bump the name.
} while (acdbEntNext(ent1, ent0) == RTNORM);
```

NOTE You can also go through the database by "bumping" a single variable in the `acdbEntNext()` call (such as `acdbEntNext(ent0, ent0)`), but if you do, the value of the variable is no longer defined once the loop ends.

The `acdbEntLast()` function retrieves the name of the last entity in the database. The last entity is the most recently created main entity, so `acdbEntLast()` can be called to obtain the name of an entity that has just been created by means of a call to `acedCommand()`, `acedCmd()`, or `acdbEntMake()`.

The `acedEntSel()` function prompts the AutoCAD user to select an entity by specifying a point on the graphics screen; `acedEntSel()` returns both the entity name and the value of the specified point. Some entity operations require knowledge of the point by which the entity was selected. Examples from the set of existing AutoCAD commands include BREAK, TRIM, EXTEND, and OSNAP.

Entity Handles and Their Uses

The `acdbHandEnt()` function retrieves the name of an entity with a specific handle. Like entity names, handles are unique within a drawing. Unlike entity names, an entity's handle is constant throughout its life. ObjectARX applications that manipulate a specific database can use `acdbHandEnt()` to obtain the current name of an entity they must use.

The following sample code fragment uses `acdbHandEnt()` to obtain an entity name and to print it out.

```
char handle[17];
ads_name e1;

strcpy(handle, "5a2");

if (acdbHandEnt(handle, e1) != RTNORM)
    acdbFail("No entity with that handle exists\n");
else
    acutPrintf("%ld", e1[0]);
```

In one particular editing session, this code might print out 60004722. In another editing session with the same drawing, it might print an entirely different number. But in both cases, the code is accessing the same entity.

The `acdbHandEnt()` function has an additional use: entities deleted from the database (with `acdbEntDel()`) are not purged until you leave the current drawing (by exiting AutoCAD or switching to another drawing). This means that `acdbHandEnt()` can recover the names of deleted entities, which can then be restored to the drawing by a second call to `acdbEntDel()`.

Entities in drawings cross-referenced with XREF Attach are not actually part of the current drawing; their handles are unchanged and cannot be accessed by `acdbHandEnt()`. 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 that each handle in the original drawing remains unique.

NOTE Extended data can include entity handles to save relational structures in a drawing. In some circumstances, these handles require translation or maintenance. See "Using Handles in Extended Data" on page 240.

Entity Context and Coordinate Transform Data

The `acedNEntSelP()` function is similar to `acedEntSel()`, except that it passes two additional result arguments to facilitate the handling of entities that are nested within block references.

NOTE Another difference between `acedNEntSelP()` and `acedEntSel()` is that when the user responds to an `acedNEntSelP()` call by selecting a complex entity, `acedNEntSelP()` returns the selected subentity and not the complex entity's header as `acedEntSel()` does. For example, when the user selects a polyline, `acedNEntSelP()` returns a vertex subentity instead of the polyline header. To retrieve the polyline header, the application must use `acdbEntNext()` to walk forward to the `Seqend` subentity and obtain the name of the header from the `Seqend` subentity's -2 group. This is true also when the user selects attributes in a nested block reference and when the pick point is specified in the `acedNEntSelP()` call.

Coordinate Transformation

The first of the additional arguments returned by `acedNEntSelP()` is a 4x4 transformation matrix of type `ads_matrix`. This matrix is known as the Model to World Transformation Matrix. It enables the application to transform points in the entity's definition data (and extended data, if that is present) from the entity's model coordinate system (MCS) into the World Coordinate System (WCS). The MCS applies only to nested entities. The origin of the MCS is the insert point of the block, and its orientation is that of the UCS that was in effect when the block was created.

If the selected entity is not a nested entity, the transformation matrix is set to the identity matrix. The transformation is expressed by the following matrix multiplication:

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1.0 \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \bullet \begin{bmatrix} X \\ Y \\ Z \\ 1.0 \end{bmatrix}$$

$$X' = M_{00}X + M_{01}Y + M_{02}Z + M_{03}$$

$$Y' = M_{10}X + M_{11}Y + M_{12}Z + M_{13}$$

$$Z' = M_{20}X + M_{21}Y + M_{22}Z + M_{23}$$

The individual coordinates of a transformed point are obtained from the equations where M_{mn} is 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. See "Transformation Matrices" on page 535.

NOTE To transform a vector rather than a point, do not add the translation vector $[M_{03}\ M_{13}\ M_{23}]$ (from the fourth column of the transformation matrix).

The following sample code defines a function, `mcs2wcs()`, that performs the transformations described by the preceding equations. It takes the transformation matrix returned by `acedNEntSelP()` and a single point (presumably from the definition data of a nested entity), and returns the translated point. If the third argument to `mcs2wcs()`, `is_pt`, is set to 0 (`FALSE`), the last column of the transformation matrix—the translation vector or displacement—is not added to the result. This enables the function to translate a vector as well as a point.

```
void mcs2wcs(xform, entpt, is_pt, worldpt)
ads_matrix xform;
ads_point entpt, worldpt;
int is_pt;
```

```
{
    int i, j;

    worldpt[X] = worldpt[Y] = worldpt[Z] = 0.0;

    for (i=X; i<=Z; i++)
        for (j=X; j<=Z; j++)
            worldpt[i] += xform[i][j] * entpt[j];

    if (is_pt) // If it's a point, add in the displacement
        for (i=X; i<=Z; i++)
            worldpt[i] += xform[i][T];
}
```

The following code fragment shows how `mcs2wcs()` might be used in conjunction with `acedNEntSelP()` to translate point values into the current WCS.

```
ads_name usrent, containent;
ads_point usrpt, defpt, wcspt;
ads_matrix matrix;
struct resbuf *containers, *data, *rb, *prevrb;

status = acedNEntSelP(NULL, usrent, usrpt, FALSE, matrix,
    &containers);
if ((status != RTNORM) || (containers == NULL))
    return BAD;
data = acdbEntGet(usrent);

// Extract a point (defpt) from the data obtained by calling
// acdbEntGet() for the selected kind of entity.
.
.
.
mcs2wcs(matrix, defpt, TRUE, wcspt);
```

The `acedNEntSelP()` function also allows the program to specify the pick point. A `pickflag` argument determines whether or not `acedNEntSelP()` is called interactively.

In the following example, the `acedNEntSelP()` call specifies its own point for picking the entity and does not prompt the user. The `pickflag` argument is `TRUE` to indicate that the call supplies its own point value (also, the `prompt` is `NULL`).

```
ads_point ownpoint;

ownpoint[X] = 2.7; ownpoint[Y] = 1.5; ownpoint[Z] = 0.0;

status = acedNEntSelP(NULL, usrent, ownpt, TRUE, matrix,
    &containers);
```

The `acedNEntSel()` function is provided for compatibility with existing ObjectARX applications. New applications should be written using `acedNEntSelP()`.

The Model to World Transformation Matrix returned by the call to `acedNEntSel()` has the same purpose as that returned by `acedNEntSelP()`, but it is a 4x3 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:

$$\begin{bmatrix} X' & Y' & Z' & 1.0 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1.0 \end{bmatrix} \bullet \begin{bmatrix} M_{00} & M_{10} & M_{20} \\ M_{01} & M_{11} & M_{21} \\ M_{02} & M_{12} & M_{22} \\ M_{03} & M_{13} & M_{23} \end{bmatrix}$$

The equations for deriving the new coordinates are as follows:

$$X' = XM_{00} + YM_{01} + ZM_{02} + M_{03}$$

$$Y' = XM_{10} + YM_{11} + ZM_{12} + M_{13}$$

$$Z' = XM_{20} + YM_{21} + ZM_{22} + M_{23}$$

Although the matrix format is different, the formulas are equivalent to those for the `ads_matrix` type, and the only change required to adapt `mcs2wcs()` for use with `acedNEntSel()` is to declare the matrix argument as an array of four points.

```
void mcs2wcs(xform, entpt, is_pt, worldpt);
ads_point xform[4]; // 4x3 version
ads_point entpt, worldpt;
int is_pt;
```

The identity form of the 4x3 matrix is as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

In addition to using a different matrix convention, `acedNEntSel()` doesn't let the program specify the pick point.

Context Data

The function `acedNEntSelP()` provides an argument for context data, `refstkres`. (This is another feature not provided by `acedEntSel()`.) The `refstkres` argument is a pointer to a linked list of result buffers that contains the names of the entity's container blocks. The list is ordered from lowest to highest. In other words, the first name in the list is the name of the block containing the selected entity, and the last name in the list is the name of the block that was directly inserted into the drawing. The following figure shows the format of this list.

If the selected entity `entres` is not a nested entity, `refstkres` is a `NULL` pointer. This is a convenient way to test whether or not the entity's coordinates need to be translated. (Because `xformres` is returned as the identity matrix for entities that are not nested, applying it to the coordinates of such entities does no harm but does cost some needless execution time.)

Using declarations from the previous `acedNEntSelP()` example, the name of the block that immediately contains the user-selected entity can be found by the following code (in the `acedNEntSelP()` call, the `pickflag` argument is `FALSE` for interactive selection).

```
status = acedNEntSelP(NULL, usrent, usrpt, FALSE, matrix,
    &containers);
if ((status != RTNORM) || (containers == NULL))
    return BAD;

containent[0] = containers->resval.rlname[0];
containent[1] = containers->resval.rlname[1];
```

The name of the outermost container (that is, the entity originally inserted into the drawing) can be found by a sequence such as the following:

```
// Check that containers is not already NULL.

rb = containers;
while (rb != NULL) {
    prevrb = rb;
    rb = containers->rbnext;
}
// The result buffer pointed to by prevrb now contains the
// name of the outermost block.
```

In the following example, the current coordinate system is the WCS. Using AutoCAD, create a block named SQUARE consisting of four lines.

Command: **line**
From point: **1,1**
To point: **3,1**
To point: **3,3**
To point: **1,3**
To point: **c**

Command: **block**
Block name (or ?): **square**
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**
Origin/ZAxis/3point/Entity/View/X/Y/Z/Prev/Restore/Save/Del/?/
<World>: **z**
Rotation angle about Z axis <0>: **45**

Command: **insert**
Block name (or ?): **square**
Insertion point: **7,0**
X scale factor <1> / Corner / XYZ: ENTER
Y scale factor (default=X): ENTER
Rotation angle: ENTER

If an ObjectARX application calls `acedNEntSelP()` (or `acedNEntSel()`) and the user selects the lower-left side of the square, these functions set the `entres` argument to equal the name of the selected line. They set the pick point (`ptres`) to (6.46616,-1.0606,0.0) or a nearby point value. They return the transformation matrix (`xformres`) as shown in the following figure. Finally, they set the list of container entities (`refstkres`) to point to a single result buffer containing the entity name of the block SQUARE.

$$\begin{bmatrix} 0.707107 & -0.707107 & 0.0 & 4.94975 \\ 0.707107 & 0.707107 & -0.0 & 4.94975 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

ads_nentselp() result

$$\begin{bmatrix} 0.707107 & 0.707107 & 0.0 \\ -0.707107 & 0.707107 & 0.0 \\ 0.0 & -0.0 & 1.0 \\ 4.94975 & 4.94975 & 0.0 \end{bmatrix}$$

ads_nentsel() result

Entity Data Functions

Some functions operate on entity data and can be used to modify the current drawing database. The `acdbEntDel()` function deletes a specified entity. The entity is not purged from the database until you leave the current drawing. So if the application calls `acdbEntDel()` a second time during that session and specifies the same entity, the entity is undeleted. (You can use `acdbHandEnt()` to retrieve the names of deleted entities.)

NOTE Using `acdbEntDel()`, attributes and polyline vertices cannot be deleted independently from their parent entities; `acdbEntDel()` operates only on main entities. To delete an attribute or vertex, use `acedCommand()` or `acedCmd()` to invoke the AutoCAD ATTEDIT or PEDIT commands, use `acdbEntMod()` to redefine the entity without the unwanted subentities, or open the vertex or attribute and use its `erase()` method to erase it.

The `acdbEntGet()` function returns the definition data of a specified entity. The data is returned as a linked list of result buffers. The type of each item (buffer) in the list is specified by a DXF group code. The first item in the list contains the entity's current name (`restype == -1`).

An ObjectARX application could retrieve and print the definition data for an entity by using the following two functions. (The `printdxf()` function does not handle extended data.)

```
void getlast()
{
    struct resbuf *ebuf, *eb;
    ads_name ent1;

    acdbEntLast(ent1);
    ebuf = acdbEntGet(ent1);

    eb = ebuf;

    acutPrintf("\nResults of entgetting last entity\n");

// Print items in the list.
    for (eb = ebuf; eb != NULL; eb = eb->rbnext)
        printdxf(eb);

// Release the acdbEntGet() list.
    acutRelRb(ebuf);
}

int printdxf(eb)
struct resbuf *eb;
{
    int rt;

    if (eb == NULL)
        return RTNONE;

    if ((eb->restype >= 0) && (eb->restype <= 9))
        rt = RTSTR ;
    else if ((eb->restype >= 10) && (eb->restype <= 19))
        rt = RT3DPOINT;
    else if ((eb->restype >= 38) && (eb->restype <= 59))
        rt = RTREAL ;
    else if ((eb->restype >= 60) && (eb->restype <= 79))
        rt = RTSHORT ;
    else if ((eb->restype >= 210) && (eb->restype <= 239))
        rt = RT3DPOINT ;
    else if (eb->restype < 0)
```

```
// Entity name (or other sentinel)
        rt = eb->restype;
    else
        rt = RTNONE;

    switch (rt) {

    case RTSHORT:
        acutPrintf("(%d . %d)\n", eb->restype,
            eb->resval.rint);
        break;

    case RTREAL:
        acutPrintf("(%d . %0.3f)\n", eb->restype,
            eb->resval.rreal);
        break;

    case RTSTR:
        acutPrintf("(%d . \"%s\")\n", eb->restype,
            eb->resval.rstring);
        break;

    case RT3DPOINT:
        acutPrintf("(%d . %0.3f %0.3f %0.3f)\n",
            eb->restype,
            eb->resval.rpoint[X], eb->resval.rpoint[Y],
            eb->resval.rpoint[Z]);
        break;

    case RTNONE:
        acutPrintf("(%d . Unknown type)\n", eb->restype);
        break;

    case -1:
    case -2:
// First block entity
        acutPrintf("(%d . <Entity name: %8lx>)\n",
            eb->restype, eb->resval.rlname[0]);
    }

    return eb->restype;
}
```

In the next example, the following (default) conditions apply to the current drawing.

- The current layer is 0
- The current linetype is CONTINUOUS
- The current elevation is 0
- Entity handles are disabled

Also, 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
```

Then a call to `getlast()` would print the following (the name value will vary).

```
Results from acdbEntGet() of last entity:
(-1 . <Entity name: 60000014>)
(0 . "LINE")
(8 . "0")
(10 1.0 2.0 0.0)
(11 6.0 6.0 0.0)
(210 0.0 0.0 1.0)
```

NOTE The `printdxf()` function prints the output in the format of an AutoLISP association list, but the items are stored in a linked list of result buffers.

The result buffer at the start of the list (with a -1 sentinel code) contains the name of the entity that this list represents. The `acdbEntMod()` function uses it to identify the entity to be modified.

The codes for the components of the entity (stored in the `restype` field) are those used by DXF. As with DXF, the entity header items are returned only if they have values other than the default. Unlike DXF, optional entity definition fields are returned regardless of whether they equal their defaults. This simplifies processing; an application can always assume that these fields are present. Also unlike DXF, associated *X*, *Y*, and *Z* coordinates are returned as a single point variable (`resval.rpoint`), not as separate *X* (10), *Y* (20), and *Z* (30) groups. The `restype` value contains the group number of the *X* coordinate (in the range 10–19).

To find a group with a specific code, an application can traverse the list. The `entitem()` function shown here searches a result buffer list for a group of a specified type.

```
static struct resbuf *entitem(rchain, gcode)
struct resbuf *rchain;
int gcode;
{
    while ((rchain != NULL) && (rchain->restype != gcode))
        rchain = rchain->rbnext;

    return rchain;
}
```

If the DXF group code specified by the `gcode` argument is not present in the list (or if `gcode` is not a valid DXF group), `entitem()` "falls off the end" and returns `NULL`. Note that `entitem()` is equivalent to the AutoLISP function `(assoc)`.

The `acdbEntMod()` function modifies an entity. It passes a list that has the same format as a list returned by `acdbEntGet()`, but with some of the entity group values (presumably) modified by the application. This function complements `acdbEntGet()`; the primary means by which an ObjectARX application updates the database is by retrieving an entity with `acdbEntGet()`, modifying its entity list, and then passing the list back to the database with `acdbEntMod()`.

NOTE To restore the default value of an entity's color or linetype, use `acdbEntMod()` to set the color to 256, which is BYLAYER, or the linetype to BYLAYER.

The following code fragment retrieves the definition data of the first entity in the drawing, and changes its layer property to MYLAYER.

```
ads_name en;
struct resbuf *ed, *cb;
char *nl = "MYLAYER";

if (acdbEntNext(NULL, en) != RTNORM)
    return BAD; // Error status

ed = acdbEntGet(en); // Retrieve entity data.
```

```
for (cb = ed; cb != NULL; cb = cb->rbnext)
    if (cb->restype == 8) {          // DXF code for Layer
        // Check to make sure string buffer is long enough.
        if (strlen(cb->resval.rstring) < (strlen(nl)))
            // Allocate a new string buffer.
            cb->resval.rstring = realloc(cb->resval.rstring,
                strlen(nl) + 1);

        strcpy(cb->resval.rstring, nl);

        if (acdbEntMod(ed) != RTNORM) {
            acutRelRb(ed);
            return BAD; // Error
        }
        break; // From the for loop
    }
acutRelRb(ed); // Release result buffer.
```

Memory management is the responsibility of an ObjectARX application. Code in the example ensures that the string buffer is the correct size, and it releases the result buffer returned by `acdbEntGet()` (and passed to `acdbEntMod()`) once the operation is completed, whether or not the call to `acdbEntMod()` succeeds.

NOTE If you use `acdbEntMod()` 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 `acdbEntDel()`.

An application can also add an entity to the drawing database by calling the `acdbEntMake()` function. Like `acdbEntMod()`, the argument to `acdbEntMake()` is a result-buffer list whose format is similar to that of a list returned by `acdbEntGet()`. (The `acdbEntMake()` call ignores the entity name field [-1] if that is present.) The new entity is appended to the drawing database (it becomes the last entity in the drawing). If the entity is a complex entity (a polyline or block), it is not appended to the database until it is complete.

The following sample code fragment creates a circle on the layer MYLAYER.

```
int status;
struct resbuf *entlist;
ads_point center = {5.0, 7.0, 0.0};
char *layer = "MYLAYER";

entlist = acutBuildList(RTDXF0, "CIRCLE",// Entity type
    8, layer, // Layer name
    10, center, // Center point
    40, 1.0, // Radius
    0 );
```

```
if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}

status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake buffer.

if (status == RTERROR) {
    acdbFail("Unable to make circle entity\n");
    return BAD;
}
```

Both `acdbEntMod()` and `acdbEntMake()` perform the same consistency checks on the entity data passed to them as the AutoCAD DXFIN command performs when reading DXF files. They fail if they cannot create valid drawing entities.

Complex Entities

A complex entity (a polyline or block) must be created by multiple calls to `acdbEntMake()`, using a separate call for each subentity. When `acdbEntMake()` 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). Each subsequent `acdbEntMake()` call appends the new subentity to the file. When the definition of the complex entity is complete (that is, when `acdbEntMake()` receives an appropriate `Seqend` or `Endblk` subentity), the entity is checked for consistency, and if valid, it is added to the drawing. The file is deleted when the complex entity is complete or when its creation is canceled.

The following example contains five calls to `acdbEntMake()` that create a single complex entity, a 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.

```
int status;
struct resbuf *entlist, result;
ads_point newpt;

entlist = acutBuildList(
    RTDXF0, "POLYLINE",// Entity type
    62, 5, // Color (blue)
    6, "dashed",// Linetype
    66, 1, // Vertices follow.
    0);

if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
```

```
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake() buffer.

if (status != RTNORM) {
    acutPrintf ("%d",status);
    acedGetVar ("ERRNO", &result);
    acutPrintf ("ERRNO == %d, result.resval.rint);
    acdbFail("Unable to start polyline\n");
    return BAD;
}

newpt[X] = 1.0;
newpt[Y] = 1.0;
newpt[Z] = 0.0; // The polyline is planar

entlist = acutBuildList(
    RTDXF0, "VERTEX", // Entity type
    62, 5, // Color (blue)
    6, "dashed", // Linetype
    10, newpt, // Start point
    0);

if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}

status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake() buffer.

if (status != RTNORM) {
    acdbFail("Unable to add polyline vertex\n");
    return BAD;
}

newpt[X] = 4.0;
newpt[Y] = 6.0;

entlist = acutBuildList(
    RTDXF0, "VERTEX", // Entity type
    62, 5, // Color (blue)
    6, "dashed", // Linetype
    10, newpt, // Second point
    0);

if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}

status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake() buffer.
```

```
if (status != RTNORM) {
    acdbFail("Unable to add polyline vertex\n");
    return BAD;
}

newpt[X] = 3.0;
newpt[Y] = 2.0;

entlist = acutBuildList(
    RTDXF0, "VERTEX", // Entity type
    62, 5, // Color (blue)
    6, "dashed", // Linetype
    10, newpt, // Third point
    0);

if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}

status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake() buffer.

if (status != RTNORM) {
    acdbFail("Unable to add polyline vertex\n");
    return BAD;
}

entlist = acutBuildList(
    RTDXF0, "SEQEND", // Sequence end
    62, 5, // Color (blue)
    6, "dashed", // Linetype
    0);

if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}

status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake() buffer.

if (status != RTNORM) {
    acdbFail("Unable to complete polyline\n");
    return BAD;
}
```

Creating a block is similar, except that when `acdbEntMake()` successfully creates the `Endblk` entity, it returns a value of `RTKWORD`. You can verify the name of the new block by a call to `acedGetInput()`.

Anonymous Blocks

You can create anonymous blocks by calls to `acdbEntMake()`. To do so, you must open the block with a name whose first character is `*` and a block type flag (group 70) whose low-order bit is set to 1. AutoCAD assigns the new anonymous block a name; characters in the name string that follow the `*` are often ignored. You then create the anonymous block the way you would create a regular block, except that it is more important to call `acedGetInput()`. Because the name is generated by AutoCAD, your program has no other way of knowing the name of the new block.

The following code begins an anonymous block, ends it, and retrieves its name.

```
int status;
struct resbuf *entlist;
ads_point basept;
char newblkname[20];

ads_point pnt1 = ( 0.0, 0.0, 0.0);
entlist = acutBuildList(
    RTDXF0, "BLOCK",
    2, "*ANON", // Only the '*' matters.
    10, "1", // No other flags are set.
    0 );

if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}

status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake buffer.

if (status != RTNORM) {
    acdbFail("Unable to start anonymous block\n");
    return BAD;
}

// Add entities to the block by more acdbEntMake calls.
.
.
.
entlist = acutBuildList(RTDXF0, "ENDBLK", 0 );

if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
```

```
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake buffer.

if (status != RTKWORD) {
    acdbFail("Unable to close anonymous block\n");
    return BAD;
}
status = acedGetInput(newblkname);

if (status != RTNORM) {
    acdbFail("Anonymous block not created\n");
    return BAD;
}
```

To reference an anonymous block, create an insert entity with `acdbEntMake()`. (You cannot pass an anonymous block to the INSERT command.)

Continuing the previous example, the following code fragment inserts the anonymous block at (0,0).

```
basept[X] = basept[Y] = basept[Z] = 0.0;

entlist = acutBuildList(
    RTDXF0, "INSERT",
    2, newblkname, // From acedGetInput
    10, basept,
    0 );

if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}

status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake buffer.

if (status != RTNORM) {
    acdbFail("Unable to insert anonymous block\n");
    return BAD;
}
```

Entity Data Functions and Graphics Screen

Changes to the drawing made by the entity data functions are reflected on the graphics screen, provided that the entity being deleted, undeleted, modified, or made is in an area and is on a layer that is currently visible. There is one exception: when `acdbEntMod()` modifies a subentity, it does not update the image of the entire (complex) entity. The reason should be clear. If, for example, an application were to modify 100 vertices of a complex polyline with 100 iterated calls to `acdbEntMod()`, the time required to recalculate and redisplay the entire polyline as each vertex was changed would be unaccept-

ably slow. Instead, an application can perform a series of subentity modifications and then redisplay the entire entity with a single call to the `acdbEntUpd()` function.

In the following example, the first entity in the current drawing is a polyline with several vertices. The following code modifies the second vertex of the polyline and then regenerates its screen image.

```
ads_name e1, e2;
struct resbuf *ed, *cb;

if (acdbEntNext(NULL, e1) != RTNORM) {
    acutPrintf("\nNo entities found.  Empty drawing.");
    return BAD;
}

acdbEntNext(e1, e2);

if ((ed = acdbEntGet(e2)) != NULL) {
    for (cb = ed; cb != NULL; cb = cb->rbnext)

        if (cb->restype == 10) { // Start point DXF code
            cb->resval.rpoint[X] = 1.0;// Change coordinates.
            cb->resval.rpoint[Y] = 2.0;

            if (acdbEntMod(ed) != RTNORM) { // Move vertex.
                acutPrintf("\nBad vertex modification.");
                acutRelRb(ed);
                return BAD;
            } else {
                acdbEntUpd(e1); // Regen the polyline.
                acutRelRb(ed);
                return GOOD; // Indicate success.
            }
        }
    acutRelRb(ed);
}
return BAD; // Indicate failure.
```

The argument to `acdbEntUpd()` can specify either a main entity or a subentity; in either case, `acdbEntUpd()` regenerates the entire entity. Although its primary use is for complex entities, `acdbEntUpd()` can regenerate any entity in the current drawing.

NOTE If the modified entity is in a block definition, then the `acdbEntUpd()` function is not sufficient. You must regenerate the drawing by invoking the AutoCAD REGEN command (with `acedCmd()` or `acedCommand()`) to ensure that all instances of the block references are updated.

Notes on Extended Data

Several ObjectARX functions are provided to handle extended data. An entity's extended data follows the entity's normal definition data. This is illustrated by the next figure, which shows the scheme of a result-buffer list for an entity containing extended data.

An entity's extended data can be retrieved by calling `acdbEntGetX()`, which is similar to `acdbEntGet()`. The `acdbEntGetX()` function retrieves an entity's normal definition data and the extended data for applications specified in the `acdbEntGetX()` call.

NOTE When extended data is retrieved with `acdbEntGetX()`, the beginning of extended data is indicated by a -3 sentinel code; the -3 sentinel is in a result buffer that precedes the first 1001 group. The 1001 group contains the application name of the first application retrieved, as shown in the figure.

Organization of Extended Data

Extended data consists of one or more 1001 groups, each of which begins with a unique application name. Application names are string values. The extended data groups returned by `acdbEntGetX()` 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 doesn't use it. Group codes for extended data are in the range 1000–1071, 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. Letters in the name are converted to uppercase.

A group of extended data cannot consist of an application name with no other data.

To delete extended data

1. Call `acdbEntGet()` to retrieve the entity.
2. Add to the end of the list returned by `acdbEntGet()` a resbuf with a `restype` of -3.
3. Add to the end of the list another resbuf with a `restype` of 1001 and a `resval.rstring` set to `<appname>`.

If you attempt to add a 1001 group but no other extended data to an existing entity, the attempt is ignored. If you attempt to make an entity whose only extended data group is a single 1001 group, the attempt fails.

Layer name	1003. Name of a layer associated with the extended data.
Database handle	1005. Handles of entities in the drawing database. Under certain conditions, AutoCAD translates these.
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 (long) integer. If the value that appears in a 1071 group is a short integer or a real value, it is converted to a long integer; if it is invalid (for example, a string), it is converted to a long zero (0L).

Control string	1002. An extended data 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.) When it reads the extended data, AutoCAD checks to ensure that braces are balanced correctly.
Binary data	1004. Binary data is organized into variable-length chunks, which can be handled in ObjectARX with the `ads_binary` structure. The maximum length of each chunk is 127 bytes.
World space position	1011. Unlike a simple 3D point, the world space coordinates are moved, scaled, rotated, and mirrored along with the parent entity to which the extended data belongs. The world space position is also stretched when the STRETCH command is applied to the parent entity and this point lies within the selection window.
World space displacement	1012. A 3D point that is scaled, rotated, or mirrored along with the parent, but not stretched or moved.
World direction	1013. Also a 3D point that is rotated, or mirrored along with the parent, but not scaled, stretched, or moved. The world direction is a normalized displacement that always has a unit length.
Distance	1041. A real value that is scaled along with the parent entity.
Scale factor	1042. Also a real value that is scaled along with the parent.

NOTE If a 1001 group appears within a list, it is treated as a string and does not begin a new application group.

Registering an Application

Application names are saved with the extended data of each entity that uses them and in the APPID table. An application must register the name or names it uses. In ObjectARX, this is done by a call to `acdbRegApp()`. The `acdbRegApp()` function specifies a string to use as an application name. It returns `RTNORM` if it can successfully add the name to APPID; otherwise, it returns `RTERROR`. A result of `RTERROR` usually indicates that the name is already in the symbol table. This is not an actual error condition but a

normally 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, because `acdbRegApp()` needs to be called only once per drawing. If the name is not there, the application must register it; otherwise, it can go ahead and use the data.

The following sample code fragment shows the typical use of `acdbRegApp()`.

```
#define APPNAME "Local_Operation_App_3-2"

struct resbuf *rbp;
static char *local_appname = APPNAME;
// The static declaration prevents a copy being made of the string
// every time it's referenced.
.
.
.
if ((rbp = acdbTblSearch("APPID", local_appname, 0)) == NULL) {
    if (acdbRegApp(APPNAME) != RTNORM) { // Some other
                                         // problem
        acutPrintf("Can't register XDATA for %s.",
                   local_appname);
        return BAD;
    }
} else {
    acutRelRb(rbp);
}
```

Retrieving Extended Data

An application can obtain registered extended data by calling the `acdbEntGetX()` function, which is similar to `acdbEntGet()`. While `acdbEntGet()` returns only definition data, `acdbEntGetX()` returns both the definition data and the extended data for the applications it requests. It requires an additional argument, `apps`, that specifies the application names (this differs from AutoLISP, in which the `(entget)` function has been extended to accept an optional argument that specifies application names). The names passed to `acdbEntGetX()` must correspond to applications registered by a previous call to `acdbRegApp()`; they can also contain wild-card characters. If the `apps` argument is a `NULL` pointer, the call to `acdbEntGetX()` is identical to an `acdbEntGet()` call.

The following sample code fragment shows a typical sequence for retrieving extended data for two specified applications. Note that the `apps` argument passes application names in linked result buffers.

```
static struct resbuf    appname2 = {NULL, RTSTR},
                        appname1 = {&appname2, RTSTR},
                        *working_ent;
```

```
strsave(appname1.rstring, "MY_APP_1");
strsave(appname2.rstring, "SOMETHING_ELSE");
.
.
.

// Only extended data from "MY_APP_1" and
// "SOMETHING_ELSE" are retrieved:
working_ent = acdbEntGetX(&work_ent_addr, &appname1);
if (working_ent == NULL) {
    // Gracefully handle this failure.
    .
    .
    .
}

// Update working entity groups.

status = acdbEntMod(working_ent);
// Only extended data from registered applications still in the
// working_ent list are modified.
```

As the sample code shows, extended data retrieved by the `acdbEntGetX()` function can be modified by a subsequent call to `acdbEntMod()`, just as `acdbEntMod()` is used to modify normal definition data. (Extended data can also be created by defining it in the entity list passed to `acdbEntMake()`.)

Returning the extended data of only specifically requested applications protects one application from damaging the data of another application. It also controls the amount of memory that an application uses, and simplifies the extended data processing that an application performs.

NOTE Because the strings passed with `apps` can include wild-card characters, an application name of "`*`" will cause `acdbEntGetX()` to return all extended data attached to an entity.

Managing Extended Data Memory Use

Extended data is limited to 16 kilobytes per entity. Because the extended data of an entity can be created and maintained by multiple applications, this can lead to problems when the size of the extended data approaches its limit. ObjectARX provides two functions, `acdbXdSize()` and `acdbXdRoom()`, to assist in managing the memory that extended data occupies. When `acdbXdSize()` is passed a result-buffer list of extended data, it returns the amount of memory (in bytes) that the data will occupy; when `acdbXdRoom()` is passed the name of an entity, it returns the remaining number of free bytes that can still be appended to the entity.

The `acdbXdSize()` function must read an extended data list, which can be large. Consequently, this function can be slow, so it is recommended that you don't call it frequently. A better approach is to use it (in conjunction with `acdbXdRoom()`) in an error handler. If a call to `acdbEntMod()` fails, you can use `acdbXdSize()` and `acdbXdRoom()` to find out whether the call failed because the entity ran out of extended data, and then take appropriate action if that is the reason for the failure.

Using 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 entity's handle in its extended data. The handle can be retrieved later and passed to `acdbHandEnt()` to obtain the other entity. Because more than one entity can reference another, extended data handles are not necessarily unique; the AUDIT command does require that handles in extended data are either `NULL` or valid entity handles (within the current drawing). The best way to ensure that extended entity handles are valid is to obtain a referenced entity's handle directly from its definition data, by means of `acdbEntGet()`. (The handle value is in group 5 or 105.)

To reference entities in other drawings (for example, entities that are attached by means of an xref), you can avoid protests from AUDIT by using extended entity strings (group 1000) rather than handles (group 1005), because the handles of cross-referenced entities either are not valid in the current drawing or 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 entity references accordingly.

NOTE When drawings are combined by means of INSERT, INSERT *, XREF Bind (XBIND), or partial DXFIN, handles are translated so that 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 (by means of the BLOCK command), the entity within the block is assigned new handles. (If the original entity is restored with OOPS, it retains its original handles.) The value of any extended data handles remains unchanged. When a block is exploded (with EXPLODE), extended data handles are translated, in a manner similar to the way they are translated when drawings are combined. If the extended data handle refers to an entity not within the block, it is unchanged; but if the extended data handle refers to an entity within the block, it is assigned the value of the new (exploded) entity's handle.

Xrecord Objects

The xrecord object is a built-in object class with a DXF name of "`XRECORD`", which stores and manages arbitrary data streams, represented externally as a result-buffer list composed of DXF groups with "normal object" groups (that is, non-xdata group codes), ranging from 1 through 369.

WARNING! The xrecord object is designed in a way that will not offend earlier versions of AutoCAD; however, the xrecord object will disappear when creating a DXF file from a pre-Release 13c4 level of AutoCAD.

Xrecord objects are generic objects intended for use by ObjectARX and AutoLISP applications. This class allows applications to create and store arbitrary object structures of arbitrary result-buffer lists of non-graphical information completely separate from entities. The root owner for all application-defined objects is either the named object dictionary, which accepts any `AcDbObject` type as an entry, including `AcDbXrecord`, or the extension dictionary of any object.

Applications are expected to use unique entry names in the named object dictionary. The logic of using a named object dictionary or extension dictionary entry name is similar to that of a REGAPP name. In fact, REGAPP names are perfect for use as entry names when appending application-defined objects to the database or a particular object.

The use of xrecord objects represents a substantial streamlining with respect to the current practice of assigning xdata to entities. Because an xrecord object does not need to be linked with an entity, you no longer need to create dummy entities (dummy entities were often used to provide more room for xdata), or entities on frozen layers.

Applications are now able to do the following:

- Protect information from indiscriminate purging or thawing of layers, which is always a threat to nongraphical information stored in xdata.
- Utilize the new object ownership/pointer reference fields (330–369) to maintain internal database object references. Arbitrary handle values are completely exempt from the object ID translation mechanics. This is opposed to 1005 xdata groups, which are translated in some cases but not in others.
- Remain unaffected by the 16K per object xdata capacity limit. This object can also be used instead of xdata on specific entities and objects, if one so wishes, with the understanding that no matter where you store xrecord objects, they have no built-in size limit, other than the limit of 2 GB imposed by signed 32-bit integer range.

In the case of object-specific state, xrecord objects are well suited for storing larger amounts of stored information, while xdata is better suited for smaller amounts of data.

When building up a hierarchy of xrecord objects (adding ownership or pointer reference to an object), that object must already exist in the database, and, thus, have a legitimate entity name. Because `acdbEntMake()` does not return an entity name, and `acdbEntLast()` only recognizes graphical objects, you must use `acdbEntMakeX()` if you are referencing nongraphical objects.

The `acdbEntMakeX()` function returns the entity name of the object added to the database (either graphical or nongraphical). The initial Release 13 implementation of `acdbEntMake()` only supported objects whose class dictated its specific owner-container object in the current drawing (such as symbol table entries, all supplied Release 13 entity types, and dictionary objects), and registered the new object with its owner. These functions will continue to do this for the same set of built-in object classes, including entities. For xrecords and all custom classes, these functions will add the object to the database, leaving it up to the application to establish its ownership links back up to the named object dictionary. The `acdbEntMakeX()` function appends the object to the database for all object types, including those that come with AutoCAD. So, even when using this function on existing entity types, your program is responsible for setting up ownership.

Symbol Table Access

The `acdbTblNext()` function sequentially scans symbol table entries, and the `acdbTblSearch()` function retrieves specific entries. Table names are specified by strings. The valid names are "`LAYER`", "`LTYPE`", "`VIEW`", "`STYLE`", "`BLOCK`", "`UCS`", "`VPORT`", and "`APPID`". Both of these functions return entries as result-buffer lists with DXF group codes.

The first call to `acdbTblNext()` returns the first entry in the specified table. Subsequent calls that specify the same table return successive entries unless the second argument to `acdbTblNext()` (`rewind`) is nonzero, in which case `acdbTblNext()` returns the first entry again.

In the following example, the function `getblock()` retrieves the first block (if any) in the current drawing, and calls the `printdxf()` function to display that block's contents in a list format.

```
void getblock()
{
    struct resbuf *bl, *rb;
```

```
    bl = acdbTblNext("BLOCK", 1); // First entry

    acutPrintf("\nResults from getblock():\n");

// Print items in the list as "assoc" items.
    for (rb = bl; rb != NULL; rb = rb->rbnext)
        printdxf(rb);

    // Release the acdbTblNext list.
    acutRelRb(bl);
}
```

Entries retrieved from the BLOCK table contain a -2 group that contains the name of the first entity in the block definition. In a drawing with a single block named BOX, a call to `getblock()` prints 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>)
```

The first argument to `acdbTblSearch()` 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, `acdbTblSearch()` returns its data. This function has a third argument, `setnext`, that can be used to coordinate operations with `acdbTblNext()`. If `setnext` is zero, the `acdbTblSearch()` call has no effect on `acdbTblNext()`, but if `setnext` is nonzero, the next call to `acdbTblNext()` returns the table entry that follows the entry found by `acdbTblSearch()`.

The `setnext` option is especially useful when dealing with the VPORT symbol table, because all viewports in a particular viewport configuration have the same name (such as *ACTIVE).

Keep in mind that if the VPORT symbol table is accessed when TILEMODE is off, changes have no visible effect until TILEMODE is turned back on. (TILEMODE is set either by the SETVAR command or by entering its name directly.) Do not confuse the VPORT symbol table with viewport entities.

To find and process each viewport in the configuration named 4VIEW, you might use the following code:

```
struct resbuf *v, *rb;

v = acdbTblSearch("VPORT", "4VIEW", 1);

while (v != NULL} {
    for (rb = v; rb != NULL; rb = rb->rbnext)
        if (rb->restype == 2)
            if (strcmp(rb->resval.rstring, "4VIEW") == 0) {
                .// Process the VPORT entry
                .
                .
                acutRelRb(v);
// Get the next table entry.
                v = acdbTblNext("VPORT", 0);
            } else {
                acutRelRb(v);
                v = NULL; // Break out of the while loop.
                break; // Break out of the for loop.
            }
}
```

Global Functions for Interacting with AutoCAD

10

In This Chapter

- AutoCAD Queries and Commands
- Getting User Input
- Conversions
- Character Type Handling
- Coordinate System Transformations
- Display Control
- Tablet Calibration
- Wild-Card Matching

The global functions described in this chapter allow your application to communicate with AutoCAD. This chapter discusses functions for registering commands with AutoCAD, handling user input, handling data conversions, and setting up external devices such as the tablet.

AutoCAD Queries and Commands

The functions described in this section access AutoCAD commands and services.

General Access

The most general of the functions that access AutoCAD are `acedCommand()` and `acedCmd()`. Like the `(command)` function in AutoLISP, these functions send commands and other input directly to the AutoCAD Command prompt.

```
int
acedCommand(int rtype, ...);
```

```
int
acedCmd(struct resbuf *rbp);
```

Unlike most other AutoCAD interaction functions, `acedCommand()` has a variable-length argument list: arguments to `acedCommand()` are treated as pairs except for `RTLE` and `RTLB`, which are needed to pass a pick point. The first of each argument pair identifies the result type of the argument that follows, and the second contains the actual data. The final argument in the list is a single argument whose value is either 0 or `RTNONE`. Typically, the first argument to `acedCommand()` is the type code `RTSTR`, and the second data argument is a string that is the name of the command to invoke. Succeeding argument pairs specify options or data that the specified command requires. The type codes in the `acedCommand()` argument list are result types.

The data arguments must correspond to the data types and values expected by that command's prompt sequence. These can 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 (as the user might enter them) or as the values themselves (that is, as integer, real, or point values). An empty string ("") is equivalent to entering a space on the keyboard.

Because of the type identifiers, the `acedCommand()` argument list is not the same as the argument list for the AutoLISP `(command)` routine. Be aware of this if you convert an AutoLISP routine into an ObjectARX application.

There are restrictions on the commands that `acedCommand()` can invoke, which are comparable to the restrictions on the AutoLISP `(command)` function.

NOTE The `acedCommand()` and `acedCmd()` functions can invoke the AutoCAD SAVE or SAVEAS command. When they do so, AutoLISP issues a `kSaveMsg` message to all other ObjectARX applications currently loaded, but not to the application that invoked SAVE. The comparable code is sent when these functions invoke NEW, OPEN, END, or QUIT from an application.

The following sample function shows a few calls to `acedCommand()`.

```
int docmd()
{
    ads_point p1;
    ads_real rad;

    if (acedCommand(RTSTR, "circle", RTSTR, "0,0", RTSTR,
        "3,3", 0) != RTNORM)
        return BAD;
    if (acedCommand(RTSTR, "setvar", RTSTR, "thickness",
        RTSHORT, 1, 0) != RTNORM)
        return BAD;

    p1[X] = 1.0; p1[Y] = 1.0; p1[Z] = 3.0;
    rad = 4.5;

    if (acedCommand(RTSTR, "circle", RT3DPOINT, p1, RTREAL,
        rad, 0) != RTNORM)
        return BAD;

    return GOOD;
}
```

Provided that AutoCAD is at the Command prompt when this function is called, AutoCAD performs the following actions:

1. Draws a circle that passes through (3.0,3.0) and whose center is at (0.0,0.0).
2. Changes the current thickness to 1.0. Note that the first call to `acedCommand()` passes the points as strings, while the second passes a `short` integer. Either method is possible.
3. Draws another (extruded) circle whose center is at (1.0,1.0,3.0) and whose radius is 4.5. This last call to `acedCommand()` uses a 3D point and a real (double-precision floating-point) value. Note that points are passed by reference, because `ads_point` is an array type.

Using acedCmd()

The `acedCmd()` function is equivalent to `acedCommand()` but passes values to AutoCAD in the form of a result-buffer list. This is useful in situations where complex logic is involved in constructing a list of AutoCAD commands. The `acutBuildList()` function is useful for constructing command lists.

The `acedCmd()` function also has the advantage that the command list can be modified at runtime rather than be fixed at compile time. Its disadvantage is that it takes slightly longer to execute. For more information, see the *ObjectARX Reference*.

The following sample code fragment causes AutoCAD to perform a REDRAW on the current graphics screen (or viewport).

```
struct resbuf *cmdlist;

cmdlist = acutBuildList(RTSTR, "redraw", 0);
if (cmdlist == NULL) {
    acdbFail("Couldn't create list\n");
    return BAD;
}

acedCmd(cmdlist);
acutRelRb(cmdlist);
```

Pausing for User Input

If an AutoCAD command is in progress and AutoCAD encounters the PAUSE symbol as an argument to `acedCommand()` or `acedCmd()`, the command is suspended to allow direct user input, including dragging. The PAUSE symbol consists of a string that contains a single backslash. This is similar to the backslash pause mechanism provided for menus.

The following call to `acedCommand()` invokes the ZOOM command and then uses the PAUSE symbol so that the user can select one of the ZOOM options.

```
result = acedCommand(RTSTR, "Zoom", RTSTR, PAUSE, RTNONE);
```

The following call starts the CIRCLE command, sets the center point as (5,5), and then pauses to let the user drag the circle's radius on the screen. When the user specifies the chosen point (or enters the chosen radius), the function resumes, drawing a line from (5,5) to (7,5).

```
result = acedCommand(RTSTR, "circle", RTSTR, "5,5",
    RTSTR, PAUSE, RTSTR, "line", RTSTR, "5,5", RTSTR,
    "7,5", RTSTR, "", 0);
```

Passing Pick Points to AutoCAD Commands

Some AutoCAD commands (such as TRIM, EXTEND, and FILLET) require users to specify a pick point as well as the entity. To pass such pairs of entity and point data by means of `acedCommand()`, you must specify the name of the entity first and enclose the pair in the RTLB and RTLE result type codes.

The following sample code fragment creates a circle centered at (5,5) and a line that extends from (1,5) to (8,5); it assumes that the circle and line are created in an empty drawing. It then uses a pick point with the TRIM com-

mand to trim the line at the circle's edge. The `acdbEntNext()` function finds the next entity in the drawing, and the `acdbEntLast()` function finds the last entity in the drawing.

```
ads_point p1;
ads_name first, last;

acedCommand(RTSTR, "Circle", RTSTR, "5,5", RTSTR, "2",
    0);
acedCommand(RTSTR, "Line", RTSTR, "1,5", RTSTR, "8,5",
    RTSTR, "", 0);
acdbEntNext(NULL, first);     // Get circle.
acdbEntLast(last);     // Get line.
// Set pick point.
p1[X] = 2.0;
p1[Y] = 5.0;
p1[Z] = 0.0;
acedCommand(RTSTR, "Trim", RTENAME, first, RTSTR, "",
    RTLB, RTENAME, last, RTPOINT, p1, RTLE,
    RTSTR, "", 0);
```

System Variables

A pair of functions, `acedGetVar()` and `acedSetVar()`, enable ObjectARX applications to inspect and change the value of AutoCAD system variables. These functions use a string to specify the variable name (in either uppercase or lowercase), and a (single) result buffer for the type and value of the variable. A result buffer is required in this case because the AutoCAD system variables come in a variety of types: integers, real values, strings, 2D points, and 3D points.

The following sample code fragment ensures that subsequent FILLET commands use a radius of at least 1.

```
struct resbuf rb, rb1;

acedGetVar("FILLETRAD", &rb);

rb1.restype = RTREAL;
rb1.resval.rreal = 1.0;
if (rb.resval.rreal < 1.0)
    if (acedSetVar("FILLETRAD", &rb1) != RTNORM)
        return BAD; // Setvar failed.
```

In this example, the result buffer is allocated as an automatic variable when it is declared in the application. The application does not have to explicitly manage the buffer's memory use as it does with dynamically allocated buffers.

If the AutoCAD system variable is a string type, `acedGetVar()` allocates space for the string. The application is responsible for freeing this space. You can do this by calling the standard C library function `free()`, as shown in the following example:

```
acedGetVar("TEXTSTYLE", &rb);
if (rb.resval.rstring != NULL)
    // Release memory acquired for string:
    free(rb.resval.rstring);
```

AutoLISP Symbols

The functions `acedGetSym()` and `acedPutSym()` let ObjectARX applications inspect and change the value of AutoLISP variables.

In the first example, the user enters the following AutoLISP expressions:

Command: **(setq testboole t)**
T
Command: **(setq teststr "HELLO, WORLD")**
"HELLO, WORLD"
Command: **(setq sset1 (ssget))**
<Selection set: 1>

Then the following sample code shows how `acedGetSym()` retrieves the new values of the symbols.

```
struct resbuf *rb;
int rc;
long sslen;

rc = acedGetSym("testboole", &rb);
if (rc == RTNORM && rb->restype == RTT)
    acutPrintf("TESTBOOLE is TRUE\n");
acutRelRb(rb);

rc = acedGetSym("teststr", &rb);
if (rc == RTNORM && rb->restype == RTSTR)
    acutPrintf("TESTSTR is %s\n", rb->resval.rstring);
acutRelRb(rb);

rc = acedGetSym("sset1", &rb);
if (rc == RTNORM && rb->restype == RTPICKS) {
    rc = acedSSLength(rb->resval.rlname, &sslen);
    acutPrintf("SSET1 contains %lu entities\n", sslen);
}
acutRelRb(rb);
```

Conversely, `acedPutSym()` can create or change the binding of AutoLISP symbols, as follows:

```
ads_point pt1;
pt1[X] = pt1[Y] = 1.4; pt1[Z] = 10.9923;
```

```
rb = acutBuildList(RTSTR, "GREETINGS", 0);
rc = acedPutSym("teststr", rb);
acedPrompt("TESTSTR has been reset\n");
acutRelRb(rb);

rb = acutBuildList(RTLB, RTSHORT, -1,
    RTSTR, "The combinations of the world",
    RTSTR, "are unstable by nature.", RTSHORT, 100,
    RT3DPOINT, pt1,
    RTLB, RTSTR, "He jests at scars",
    RTSTR, "that never felt a wound.", RTLE, RTLE, 0);

rc = acedPutSym("longlist", rb);
acedPrompt("LONGLIST has been created\n");
acutRelRb(rb);
```

To set an AutoLISP variable to nil, make the following assignment and function call:

```
rb->restype = RTNIL;
acedPutSym("var1", rb);
```

Users can retrieve these new values. (As shown in the example, your program should notify users of any changes.)

TESTSTR has been reset.
LONGLIST has been created.
Command: **!teststr**
("GREETINGS")
Command: **!longlist**
((-1 "The combinations of the world" "are unstable by nature." 100 (1.4 1.4 10.9923) ("He jests at scars" "that never felt a wound.")))

File Search

The `acedFindFile()` function enables an application to search for a file of a particular name. The application can specify the directory to search, or it can use the current AutoCAD library path.

In the following sample code fragment, `acedFindFile()` searches for the requested file name according to the AutoCAD library path.

```
char *refname = "refc.dwg";
char fullpath[100];
.
.
.
if (acedFindFile(refname, fullpath) != RTNORM) {
    acutPrintf("Could not find file %s.\n", refname);
    return BAD;
```

If the call to `acedFindFile()` is successful, the fullpath argument is set to a fully qualified path name string, such as the following:

/home/work/ref/refc.dwg

You can also prompt users to enter a file name by means of the standard AutoCAD file dialog box. To display the file dialog box, call `acedGetFileD()`.

The following sample code fragment uses the file dialog box to prompt users for the name of an ObjectARX application.

```
struct resbuf *result;
int rc, flags;

if (result = acutNewRb(RTSTR) == NULL) {
    acdbFail("Unable to allocate buffer\n");
    return BAD;
}

result->resval.rstring=NULL;

flags = 2; // Disable the "Type it" button.

rc = acedGetFileD("Get ObjectARX Application", // Title
    "/home/work/ref/myapp",   // Default pathname
    NULL, // The default extension: NULL means "*".
    flags, // The control flags
    result); // The path selected by the user.
if (rc == RTNORM)
    rc = acedArxLoad(result->resval.rstring);
```

Object Snap

The `acedOsnap()` function finds a point by using one of the AutoCAD Object Snap modes. The snap modes are specified in a string argument.

In the following example, the call to `acedOsnap()` looks for the midpoint of a line near `pt1`.

```
acedOsnap(pt1, "midp", pt2);
```

The following call looks for either the midpoint or endpoint of a line, or the center of an arc or circle—whichever is nearest `pt1`.

```
acedOsnap(pt1, "midp,endp,center", pt2);
```

The third argument (`pt2` in the examples) is set to the snap point if one is found. The `acedOsnap()` function returns `RTNORM` if a point is found.

NOTE The APERTURE system variable determines the allowable proximity of a selected point to an entity when using Object Snap.

Viewport Descriptors

The function `acedVports()`, like the AutoLISP function `(vports)`, gets a list of descriptors of the current viewports and their locations.

The following sample code gets the current viewport configuration and passes it back to AutoLISP for display.

```
struct resbuf *rb;
int rc;

rc = acedVports(&rb);
acedRetList(rb);
acutRelRb(rb);
```

For example, given a single-viewport configuration with TILEMODE turned on, the preceding code may return the list shown in the following figure.

Similarly, if four equal-sized viewports are located in the four corners of the screen and TILEMODE is turned on, the preceding code may return the configuration shown in the next figure.

The current viewport's descriptor is always first in the list. In the list shown in the preceding figure, viewport number 5 is the current viewport.

Geometric Utilities

One group of functions enables applications to obtain geometric information. The `acutDistance()` function finds the distance between two points, `acutAngle()` finds the angle between a line and the *X* axis of the current UCS (in the *XY* plane), and `acutPolar()` finds a point by means of polar coordinates (relative to an initial point). Unlike most ObjectARX functions, these

functions do not return a status value. The `acdbInters()` function finds the intersection of two lines; it returns `RTNORM` if it finds a point that matches the specification.

NOTE Unlike `acedOsnap()`, the functions in this group simply calculate the point, line, or angle values, and do not actually query the current drawing.

The following sample code fragment shows some simple calls to the geometric utility functions.

```
ads_point pt1, pt2;
ads_point base, endpt;
ads_real rads, length;
.
.     // Initialize pt1 and pt2.
.
// Return the angle in the XY plane of the current UCS, in radians.
rads = acutAngle(pt1, pt2);
// Return distance in 3D space.
length = acutDistance(pt1, pt2);
base[X] = 1.0; base[Y] = 7.0; base[Z] = 0.0;
acutPolar(base, rads, length, endpt);
```

The call to `acutPolar()` sets `endpt` to a point that is the same distance from (1,7) as `pt1` is from `pt2`, and that is at the same angle from the *X* axis as the angle between `pt1` and `pt2`.

The Text Box Utility Function

The function `acedTextBox()` finds the diagonal coordinates of a box that encloses a text entity. The function takes an argument, `ent`, that must specify a text definition or a string group in the form of a result-buffer list. The `acedTextBox()` function sets its `p1` argument to the minimum *XY* coordinates of the box and its `p2` argument to the maximum *XY* coordinates.

If the text is horizontal and is not rotated, `p1` (the bottom-left corner) and `p2` (the top-right corner) describe the bounding box of the text. The coordinates are expressed in the Entity Coordinate System (ECS) of `ent` with the origin (0,0) at the left endpoint of the baseline. (The origin is not the bottom-left corner if the text contains letters with descenders, such as g and p.) For example, the following figure shows the results of applying `acedTextBox()` to a text entity with a height of 1.0. The figure also shows the baseline and origin of the text.

The next figure shows the point values that `acedTextBox()` returns for samples of vertical and aligned text. In both samples, the height of the letters was entered as 1.0. (For the rotated text, this height is scaled to fit the alignment points.)

Note that with vertical text styles, the points are still returned in left-to-right, bottom-to-top order, so the first point list contains negative offsets from the text origin.

The `acedTextBox()` function can also measure strings in attdef and attrib entities. For an attdef, `acedTextBox()` measures the tag string (group 2); for an attrib entity, it measures the current value (group 1).

The following function, which uses some entity handling functions, prompts the user to select a text entity, and then draws a bounding box around the text from the coordinates returned by `acedTextBox()`.

NOTE The sample `tbox()` function works correctly only if you are currently in the World Coordinate System (WCS). If you are not, the code should convert the ECS points retrieved from the entity into the UCS coordinates used by `acedCommand()`. See "Coordinate System Transformations" on page 271.

```
int tbox()
{
    ads_name tname;
    struct resbuf *textent, *tent;
    ads_point origin, lowleft, upright, p1, p2, p3, p4;
    ads_real rotatn;
    char rotatstr[15];
    if (acedEntSel("\nSelect text: ", tname, p1) != RTNORM) {
        acdbFail("No Text entity selected\n");
        return BAD;
    }
    textent = acdbEntGet(tname);
    if (textent == NULL) {
        acdbFail("Couldn't retrieve Text entity\n");
        return BAD;
    }

    tent = entitem(textent, 10);
    origin[X] = tent->resval.rpoint[X]; //ECS coordinates
    origin[Y] = tent->resval.rpoint[Y];

    tent = entitem(textent, 50);
    rotatn = tent->resval.rreal;
             // acdbAngToS() converts from radians to degrees.
    if (acdbAngToS(rotatn, 0, 8, rotatstr) != RTNORM) {
        acdbFail("Couldn't retrieve or convert angle\n");
        acutRelRb(textent);
        return BAD;
    }

    if (acedTextBox(textent, lowleft, upright) != RTNORM) {
        acdbFail("Couldn't retrieve text box
                    coordinates\n");
        acutRelRb(textent);
        return BAD;
    }
    acutRelRb(textent);

        // If not currently in the WCS, at this point add
        // acedTrans() calls to convert the coordinates
        // retrieved from acedTextBox().

    p1[X] = origin[X] + lowleft[X]; // UCS coordinates
    p1[Y] = origin[Y] + lowleft[Y];

    p2[X] = origin[X] + upright[X];
    p2[Y] = origin[Y] + lowleft[Y];

    p3[X] = origin[X] + upright[X];
    p3[Y] = origin[Y] + upright[Y];

    p4[X] = origin[X] + lowleft[X];
    p4[Y] = origin[Y] + upright[Y];
```

```
    if (acedCommand(RTSTR, "pline", RTPOINT, p1,
        RTPOINT, p2, RTPOINT, p3,RTPOINT, p4, RTSTR, "c",
        0) != RTNORM) {
            acdbFail("Problem creating polyline\n");
            return BAD;
    }
    if (acedCommand(RTSTR, "rotate", RTSTR, "L", RTSTR, "",
        RTPOINT, origin, RTSTR, rotatstr, 0) != RTNORM) {
            acdbFail("Problem rotating polyline\n");
            return BAD;
    }
    return GOOD;
}
```

The preceding example "cheats" by using the AutoCAD ROTATE command to cause the rotation. A more direct way to do this is to incorporate the rotation into the calculation of the box points, as follows:

```
ads_real srot, crot;

tent = entitem(textent, 50);
rotatn = tent->resval.rreal;
srot = sin(rotatn);
crot = cos(rotatn);
        .
        .
        .
p1[X] = origin[X] + (lowleft[X]*crot - lowleft[Y]*srot);
p1[Y] = origin[Y] + (lowleft[X]*srot + lowleft[Y]*crot);

p2[X] = origin[X] + (upright[X]*crot - lowleft[Y]*srot);
p2[Y] = origin[Y] + (upright[X]*srot + lowleft[Y]*crot);

p3[X] = origin[X] + (upright[X]*crot - upright[Y]*srot);
p3[Y] = origin[Y] + (upright[X]*srot + upright[Y]*crot);

p4[X] = origin[X] + (lowleft[X]*crot - upright[Y]*srot);
p4[Y] = origin[Y] + (lowleft[X]*srot + upright[Y]*crot);
```

Getting User Input

Several global functions enable an ObjectARX application to request data interactively from the AutoCAD user.

User-Input Functions

The user-input or `acedGetxxx()` functions pause for the user to enter data of the indicated type, and return the value in a result argument. The application can specify an optional prompt to display before the function pauses.

NOTE Several functions have similar names but are not part of the user-input group: `acedGetFunCode()`, `acedGetArgs()`, `acedGetVar()`, and `acedGetInput()`.

The following functions behave like user-input functions: `acedEntSel()`, `acedNEntSelP()`, `acedNEntSel()`, and `acedDragGen()`.

The following table briefly describes the user-input functions.

User-input function summary

Function Name	Description
acedGetInt	Gets an integer value
acedGetReal	Gets a real value
acedGetDist	Gets a distance
acedGetAngle	Gets an angle (oriented to 0 degrees as specified by the ANGBASE variable)
acedGetOrient	Gets an angle (oriented to 0 degrees at the right)
acedGetPoint	Gets a point
acedGetCorner	Gets the corner of a rectangle
acedGetKword	Gets a keyword (see the description of keywords later in this section)
acedGetString	Gets a string

With some user-input functions such as `acedGetString()`, the user enters a value on the AutoCAD prompt line. With others such as `acedGetDist()`, the user either enters a response on the prompt line or specifies the value by selecting points on the graphics screen.

If the screen is used to specify a value, AutoCAD displays rubber-band lines, which are subject to application control. A prior call to `acedInitGet()` can cause AutoCAD to highlight the rubber-band line (or box).

The `acedGetKword()` function retrieves a keyword. Keywords are also string values, but they contain no white space, can be abbreviated, and must be set up before the `acedGetKword()` call by a call to `acedInitGet()`. All user-input functions (except `acedGetString()`) can accept keyword values in addition to the values they normally return, provided `acedInitGet()` has been called to set up the keywords. User-input functions that accept keywords can also accept arbitrary text (with no spaces).

NOTE You can also use `acedInitGet()` to enable `acedEntSel()`, `acedNEntSelP()`, and `acedNEntSel()` to accept keyword input. The `acedDragGen()` function also recognizes keywords.

The AutoCAD user cannot respond to a user-input function by entering an AutoLISP expression.

The user-input functions take advantage of the error-checking capability of AutoCAD. Trivial errors (such as entering only a single number in response to `acedGetPoint()`) are trapped by AutoCAD and are not returned by the user-input function. The application needs only to check for the conditions shown in the following table.

Return values for user-input functions

Code	Description
RTNORM	User entered a valid value
RTERROR	The function call failed
RTCAN	User entered ESC
RTNONE	User entered only ENTER
RTREJ	AutoCAD rejected the request as invalid
RTKWORD	User entered a keyword or arbitrary text

The RTCAN case enables the user to cancel the application's request by pressing ESC. This helps the application conform to the style of built-in AutoCAD commands, which always allow user cancellation. The return values RTNONE and RTKWORD are governed by the function acedInitGet(): a user-input function returns RTNONE or RTKWORD only if these values have been explicitly enabled by a prior acedInitGet() call.

Control of User-Input Function Conditions

The function acedInitGet() has two arguments: val and kwl. The val argument specifies one or more control bits that enable or disable certain input values to the following acedGetxxx() call. The kwl (for keyword list) argument can specify the keywords that the functions acedGetxxx(), acedEntSel(), acedNEntSelP(), acedNEntSel(), or acedDragGen() recognize.

NOTE The control bits and keywords established by acedInitGet() apply only to the next user-input function call. They are discarded immediately afterward. The application doesn't have to call acedInitGet() a second time to clear any special conditions.

Input Options for User-Input Functions

The following table summarizes the control bits that can be specified by the val argument. To set more than one condition at a time, add the values together to create a val value between 0 and 127. If val is set to zero, none of the control conditions apply to the next user-input function call.

NOTE Future versions of AutoCAD or ObjectARX may define additional acedInitGet() control bits, so you should avoid setting any bits that are not shown in the table or described in this section.

Input options set by acedInitGet()

Code	Bit Value	Description
RSG_NONULL	1	Disallow null input
RSG_NOZERO	2	Disallow zero values
RSG_NONEG	4	Disallow negative values

Input options set by acedInitGet() (*continued*)		
Code	**Bit Value**	**Description**
RSG_NOLIM	8	Do not check drawing limits, even if LIMCHECK is on
RSG_DASH	32	Use dashed lines when drawing rubber-band line or box
RSG_2D	64	Ignore *Z* coordinate of 3D points (acedGetDist() only)
RSG_OTHER	128	Allow arbitrary input—whatever the user enters

The following program excerpt shows the use of `acedInitGet()` to set up a call to the `acedGetInt()` function.

```
int age;
acedInitGet(RSG_NONULL | RSG_NOZERO | RSG_NONEG, NULL);
acedGetInt("How old are you? ", &age);
```

This sequence asks the user's age. AutoCAD automatically displays an error message and repeats the prompt if the user tries to enter a negative or zero value, press ENTER only, or enter a keyword. (AutoCAD itself rejects attempts to enter a value that is not an integer.)

The `RSG_OTHER` option lets the next user-input function call accept arbitrary input. If `RSG_OTHER` is set and the user enters an unrecognized value, the `acedGetxxx()` function returns `RTKWORD`, and the input can be retrieved by a call to `acedGetInput()`. Because spaces end user input just as ENTER does, the arbitrary input never contains a space. The `RSG_OTHER` option has the lowest priority of all the options listed in the preceding table; if the `acedInitGet()` call has disallowed negative numbers with `RSG_NONEG`, for example, AutoCAD still rejects these.

The following code allows arbitrary input (the error checking is minimal).

```
int age, rc;
char userstring[511];

acedInitGet(RSG_NONULL | RSG_NOZERO | RSG_NONEG | RSG_OTHER,
    "Mine Yours");
if ((rc = acedGetInt("How old are you? ", &age))
    == RTKWORD)
// Keyword or arbitrary input
    acedGetInput(userstring);
}
```

In this example, `acedGetInt()` returns the values shown in the following table, depending on the user's input.

Arbitrary user input

User Input	Result
41	acedGetInt() returns RTNORM and sets age to 41
m	acedGetInt() returns RTKWORD, and acedGetInput() returns "Mine"
y	acedGetInt() returns RTKWORD, and acedGetInput() returns "Yours"
twenty	acedGetInt() returns RTKWORD, and acedGetInput() returns "twenty"
what???	acedGetInt() returns RTKWORD, and acedGetInput() returns "what???"
-10	AutoCAD rejects this input and redisplays the prompt, as RSG_NONEG is set (other bit codes take precedence over RSG_OTHER)
-34.5	acedGetInt() returns RTKWORD, and acedGetInput() returns "-34.5" AutoCAD doesn't reject this value, because it expects an integer, not a real value (if this were an acedGetReal() call, AutoCAD would accept the negative integer as arbitrary input but would reject the negative real value)

NOTE The `acedDragGen()` function indicates arbitrary input (if this has been enabled by a prior `acedInitGet()` call) by returning `RTSTR` instead of `RTKWORD`.

Keyword Specifications

The optional `kwl` argument specifies a list of keywords that will be recognized by the next user-input (`acedGetxxx()`) function call. The keyword value that the user enters can be retrieved by a subsequent call to `acedGetInput()`. (The keyword value will be available if the user-input function was `acedGetKword()`.) The meanings of the keywords and the action to perform for each is the responsibility of the ObjectARX application.

The `acedGetInput()` function always returns the keyword as it appears in the `kwl` argument, with the same capitalization (but not with the optional characters, if those are specified after a comma). Regardless of how the user enters

a keyword, the application has to do only one string comparison to identify it, as demonstrated in the following example. The code segment that follows shows a call to `acedGetReal()` preceded by a call to `acedInitGet()` that specifies two keywords. The application checks for these keywords and sets the input value accordingly.

```
int stat;
ads_real x, pi = 3.14159265;
char kw[20];

// Null input is not allowed.
acedInitGet(RSG_NONULL, "Pi Two-pi");

if ((stat = acedGetReal("Pi/Two-pi/<number>: ", &x)) < 0) {
    if (stat == RTKWORD && acedGetInput(kw) == RTNORM) {
        if (strcmp(kw, "Pi") == 0) {
            x = pi;
            stat = RTNORM;
        } else if (strcmp(kw, "Two-pi") == 0) {
            x = pi * 2;
            stat = RTNORM;
        }
    }
}
if (stat != RTNORM)
    acutPrintf("Error on acedGetReal() input.\n");
else
    acutPrintf("You entered %f\n", x);
```

The call to `acedInitGet()` prevents null input and specifies two keywords: "Pi" and "Two-pi". When `acedGetReal()` is called, the user responds to the prompt Pi/Two-pi/<number> by entering either a real value (stored in the local variable `x`) or one of the keywords. If the user enters a keyword, `acedGetReal()` returns `RTKWORD`. The application retrieves the keyword by calling `acedGetInput()` (note that it checks the error status of this function), and then sets the value of `x` to pi or 2pi, depending on which keyword was entered. In this example, the user can enter either **p** to select pi or **t** to select 2pi.

Graphically Dragging Selection Sets

The function `acedDragGen()` prompts the user to drag a group of selected objects, as shown in the following example:

```
int rc;
ads_name ssname;
ads_point return_pt;

// Prompt the user for a general entity selection.
```

```
if (acedSSGet(NULL, NULL, NULL, NULL, ssname) == RTNORM)
// The newly selected entities
    rc = acedDragGen(ssname,
        "Drag selected objects", // Prompt
        0, // Display normal cursor (crosshairs)
        dragsample, // Transformation function
        return_pt); // Set to the specified location.
```

The fourth argument points to a function that does the entity transformation. See "Transformation of Selection Sets" on page 211 for examples of `dragsample()` and `acedDragGen()`.

User Breaks

The user-input functions and the `acedCommand()`, `acedCmd()`, `acedEntSel()`, `acedNEntSelP()`, `acedNEntSel()`, `acedDragGen()`, and `acedSSGet()` functions return `RTCAN` if the AutoCAD user responds by pressing ESC. An external function should treat this response as a cancel request and return immediately. ObjectARX also provides a function, `acedUsrBrk()`, that explicitly checks whether the user pressed ESC. This function enables ObjectARX applications to check for a user interrupt.

An application doesn't need to call `acedUsrBrk()` unless it performs lengthy computation between interactions with the user. The function `acedUsrBrk()` should never be used as a substitute for checking the value returned by user-input functions that can return `RTCAN`.

In some cases, an application will want to ignore the user's cancellation request. If this is the case, it should call `acedUsrBrk()` to clear the request; otherwise, the ESC will still be outstanding and will cause the next user-input call to fail. (If an application ignores the ESC, it should print a message to tell the user it is doing so.) Whenever an ObjectARX application is invoked, the ESC condition is automatically cleared.

For example, the following code fragment fails if the user enters ESC at the prompt.

```
int test()
{
    int i;

    while (!acedUsrBrk()) {
        acedGetInt("\nInput integer:", &i); // WRONG
        .
        .
        .
    }
}
```

The slightly modified code fragment that follows correctly handles an input of ESC without calling `acedUsrBrk()`.

```
int test()
{
    int i;

    for (;;) {
        if (acedGetInt("\nInput integer:", &i) != RTNORM)
            break;
        ...
    }
}
```

The following sample changes the loop condition. This construction also works correctly.

```
int test()
{
    int i;

    while (acedGetInt("\nInput integer:", &i) == RTNORM) {
        ...
    }
}
```

A valid place to use `acedUsrBrk()` is in a lengthy operation. For example, code that steps through every entity in the drawing database can be time consuming and should call `acedUsrBrk()`.

Returning Values to AutoLISP Functions

ObjectARX provides a set of functions that enables an external function to return values to AutoLISP. These value-return functions have no AutoLISP counterparts. The following table summarizes these functions.

Value-return function summary

Function Name	Returns
acedRetInt	An integer value
acedRetReal	A real value
acedRetPoint	A 3D point
acedRetStr	A string
acedRetVal	A value passed “generically” in a result buffer

Value-return function summary (*continued*)	
Function Name	**Returns**
acedRetName	An entity (RTENAME) or selection set (RTPICKS) name (see chapter 3 for more information on selection sets and entities)
acedRetT	The AutoLISP value t (true)
acedRetNil	The AutoLISP value nil
acedRetVoid	A blank value: AutoCAD doesn't display the result
acedRetList	A list of result buffers returned to AutoLISP

The following example shows the scheme of a function called when the application receives a `kInvkSubrMsg` request. It returns a real value to AutoLISP.

```
int dofun()
{
    ads_real x

// Check the arguments and input conditions here.
// Calculate the value of x.

    acedRetReal(x);
    return GOOD;
}
```

NOTE An external function can make more than one call to value-return functions upon a single `kInvkSubrMsg` request, but the AutoLISP function returns only the value passed it by the last value-return function invoked.

Conversions

The functions described in this section are utilities for converting data types and units.

String Conversions

The functions `acdbRToS()` and `acdbAngToS()` convert values used in AutoCAD to string values that can be used in output or as textual data. The `acdbRToS()` function converts a real value, and `acdbAngToS()` 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 DIMZIN dimensioning variable controls how leading and trailing zeros are written to the result string. The complementary functions `acdbDisToF()` and `acdbAngToF()` convert strings back into real (distance) values or angles. If passed a string generated by `acdbRToS()` or `acdbAngToS()`, `acdbDisToF()` and `acdbAngToF()` (respectively) are guaranteed to return a valid value.

For example, the following fragment shows calls to `acdbRToS()`. (Error checking is not shown but should be included in applications.)

```
ads_real x = 17.5;
char fmtval[12];

//Precision is the 3rd argument: 4 places in the first
// call, 2 places in the others.

acdbRToS(x, 1, 4, fmtval); // Mode 1 = scientific
acutPrintf("Value formatted as %s\n", fmtval);

acdbRToS(x, 2, 2, fmtval); // Mode 2 = decimal
acutPrintf("Value formatted as %s\n", fmtval);

acdbRToS(x, 3, 2, fmtval); // Mode 3 = engineering
acutPrintf("Value formatted as %s\n", fmtval);

acdbRToS(x, 4, 2, fmtval); // Mode 4 = architectural
acutPrintf("Value formatted as %s\n", fmtval);

acdbRToS(x, 5, 2, fmtval); // Mode 5 = fractional
acutPrintf("Value formatted as %s\n", fmtval);
```

These calls (assuming that the DIMZIN variable equals 0) display the following values on the AutoCAD text screen.

Value formatted as 1.7500E+01
Value formatted as 17.50
Value formatted as 1′-5.50″
Value formatted as 1′-5 1/2″
Value formatted as 17 1/2

When the UNITMODE system variable is set to 1, which specifies that units are displayed as entered, the string returned by `acdbRToS()` 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

The `acdbDisToF()` function complements `acdbRToS()`, so the following calls, which use the strings generated in the previous examples, all set `result` to the same value, 17.5. (Again, the examples do not show error checking.)

```
acdbDisToF("1.7500E+01", 1, &result); // 1 = scientific

acdbDisToF("17.50", 2, &result); // 2 = decimal

// Note the backslashes. Needed for inches.

acdbDisToF("1'-5.50\"", 3, &result); // 3 = engineering
acdbDisToF("1'-5 1/2\"", 4, &result); // 4 = architectural
acdbDisToF("17 1/2", 5, &result); // 5 = fractional
```

The following fragment shows calls to `acdbAngToS()` that are similar to the previous `acdbRToS()` examples.

```
ads_real ang = 3.14159;
char fmtval[12];

// Precision is the 3rd argument: 0 places in the first
// call, 4 places in the next 3, 2 in the last.

acdbAngToS(ang, 0, 0, fmtval); // Mode 0 = degrees
acutPrintf("Angle formatted as %s\n", fmtval);

acdbAngToS(ang, 1, 4, fmtval); // Mode 1 = deg/min/sec
acutPrintf("Angle formatted as %s\n", fmtval);

acdbAngToS(ang, 2, 4, fmtval); // Mode 2 = grads
acutPrintf("Angle formatted as %s\n", fmtval);

acdbAngToS(ang, 3, 4, fmtval); // Mode 3 = radians
acutPrintf("Angle formatted as %s\n", fmtval);

acdbAngToS(ang, 4, 2, fmtval); // Mode 4 = surveyor's
acutPrintf("Angle formatted as %s\n", fmtval);
```

These calls (still assuming that DIMZIN equals 0) display the following values on the AutoCAD text screen.

Angle formatted as 180
Angle formatted as 180d0′0″
Angle formatted as 200.0000g
Angle formatted as 3.1416r
Angle formatted as W

NOTE The UNITMODE system variable also affects strings returned by `acdbAngToS()` 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 `acdbAngToF()` function complements `acdbAngToS()`, so the following calls all set the result argument to the same value, 3.14159. (This is rounded up to 3.1416 in the example that uses radians.)

```
acdbAngToF("180", 0, &result); // 0 = degrees

acdbAngToF("180d0'0\"", 1, &result); // 1 = deg/min/sec

acdbAngToF("200.0000g", 2, &result); // 2 = grads

acdbAngToF("3.1416r", 3, &result); // 3 = radians

acdbAngToF("W", 4, &result); // 4 = surveyor's
```

NOTE When you have a string that specifies an angle in degrees, minutes, and seconds, you must use a backslash (\) to escape the seconds symbol (") so that it doesn't appear to be the end of the string. The second of the preceding `acdbAngToF()` examples demonstrates this.

Real-World Units

The file ac*ad.unt* defines a variety of conversions between real-world units such as miles/kilometers, Fahrenheit/Celsius, and so on. The function `acutCvUnit()` 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 that must match one of the definitions in *acad.unt*.

If the current drawing units are engineering or architectural (feet and inches), the following fragment converts a user-specified distance into meters.

```
ads_real eng_len, metric_len;
char *prmpt = "Select a distance: ";

if (acedGetDist(NULL, prmpt, &eng_len) != RTNORM)
    return BAD;
acutCvUnit(eng_len, "inches", "meters", &metric_len);
```

The `acutCvUnit()` function will not convert incompatible units, such as inches into years.

Character Type Handling

ObjectARX provides a package of character-handling functions, as shown in the table that follows. The advantage of this package over the standard C library package, *ctype.h*, is that these functions are independent of any specific character set and are not bound to ASCII. They are customized to the current AutoCAD language configuration. In other respects, they behave like their standard C counterparts.

Character type functions

Function Name	Purpose
acutIsAlpha	Verifies that the character is alphabetic
acutIsUpper	Verifies that the character is uppercase
acutIsLower	Verifies that the character is lowercase
acutIsDigit	Verifies that the character is a digit
acutIsXDigit	Verifies that the character is a hexadecimal digit
acutIsSpace	Verifies that the character is a white-space character
acutIsPunct	Verifies that the character is a punctuation character
acutIsAlNum	Verifies that the character is alphanumeric
acutIsPrint	Verifies that the character is printable
acutIsGraph	Verifies that the character is graphical
acutIsCntrl	Verifies that the character is a control character
acutToUpper	Converts the character to uppercase
acutToLower	Converts the character to lowercase

The following code fragment takes a character (the value in this example is arbitrary) and converts it to uppercase. The `acutToUpper()` function has no effect if the character is already uppercase.

```
int cc = 0x24;

cc = acutToUpper(cc);
```

Coordinate System Transformations

The `acedTrans()` 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 three-dimensional point or a three-dimensional displacement vector. This is controlled by an argument called `disp`, which must be nonzero if `pt` is treated as a displacement vector; otherwise, `pt` is treated as a point. The translated point or vector is returned in a call-by-reference result argument, which, like `pt`, is of type `ads_point`.

The arguments that specify the two coordinate systems, `from` and `to`, are both result buffers. The `from` argument specifies the coordinate system in which `pt` is expressed, and the `to` argument specifies the coordinate system of the result. Both the `from` and `to` arguments can specify a coordinate system in any of the following ways:

- An integer code (`restype == RTSHORT`) that specifies the WCS, current UCS, or current DCS (of either the current viewport or paper space).
- An entity name (`restype == RTENAME`), as returned by one of the entity name or selection set functions. This specifies the ECS of the named entity. For planar entities, the ECS can differ from the WCS. If the ECS does not differ, conversion between ECS and WCS is an identity operation.
- A 3D extrusion vector (`restype == RT3DPOINT`), which is another method of specifying an entity's ECS. Extrusion vectors are always represented in world coordinates; an extrusion vector of (0,0,1) specifies the WCS itself.

The following are descriptions of the AutoCAD coordinate systems that 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. 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, ECS, or DCS to AutoCAD commands, you must first convert them to the UCS by calling `acedTrans()`.

ECS

Entity Coordinate System. Point values returned by `acdbEntGet()` are expressed in this coordinate system relative to the entity itself. Such points are useless until they are converted into the WCS, current UCS, or current DCS, according to the intended use of the entity. Conversely, points must be translated into an ECS before they are written to the database by means of `acdbEntMod()` or `acdbEntMake()`.

DCS

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 TARGET system variable, and its *Z* 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 appears to the AutoCAD user.

When the `from` and `to` 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 (paper space or model space), and the other argument is assumed to indicate a coordinate system in the current space.

PSDCS

Paper Space DCS. This coordinate system can be transformed only to or from the DCS of the currently active model space viewport. This is essentially a 2D transformation, where the *X* and *Y* coordinates are always scaled and are offset if the `disp` argument is 0. The *Z* coordinate is scaled but is never translated; 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 `from` argument equals 3, the `to` argument must equal 2, and vice versa.

The following example translates a point from the WCS into the current UCS.

```
ads_point pt, result;
struct resbuf fromrb, torb;

pt[X] = 1.0;
pt[Y] = 2.0;
pt[Z] = 3.0;

fromrb.restype = RTSHORT;
fromrb.resval.rint = 0; // WCS

torb.restype = RTSHORT;
torb.resval.rint = 1; // UCS

// disp == 0 indicates that pt is a point:
acedTrans(pt, &fromrb, &torb, FALSE, result);
```

If the current UCS is rotated 90 degrees counterclockwise around the world *Z* axis, the call to `acedTrans()` sets the result to the point (2.0,-1.0,3.0). However, if `acedTrans()` is called as shown in the following example, the result is (-2.0,1.0,3.0).

```
acedTrans(pt, &torb, &fromrb, FALSE, result);
```

Display Control

ObjectARX has several functions for controlling the AutoCAD display, including both text and graphics screens.

Interactive Output

The basic output functions are `acedPrompt()`, which displays a message on the AutoCAD prompt line, and `acutPrintf()`, which displays text on the text screen. The `acutPrintf()` function's calling sequence is equivalent to the standard C library function `printf()`. It is provided as a separate function, because on some platforms the standard C `printf()` causes the output message to mangle the AutoCAD graphics screen. (Remember that the `acdbFail()` function also displays messages on the text screen.)

The size of a string displayed by `acedPrompt()` should not exceed the length of the graphics screen's prompt line; typically this is no more than 80 characters. The size of a string displayed by `acutPrintf()` must not exceed 132 characters, because this is the size of the string buffer used by the `acutPrintf()` function (133 bytes, with the last byte reserved for the null character).

The `acedMenuCmd()` function provides control of the display of the graphics screen menu. The `acedMenuCmd()` function activates one of the submenus of the current menu. It takes a string argument, `str`, that consists of two parts, separated by an equal sign, in the form:

```
"section=submenu"
```

where `section` indicates the menu section and `submenu` indicates which submenu to activate within that section.

For example, the following function call causes the OSNAP submenu defined in the current menu file to appear on the screen.

```
acedMenuCmd("S=OSNAP");
```

In a similar way, the following function call assigns the submenu MY-BUTTONS to the BUTTONS menu, and activates it.

```
acedMenuCmd("B=MY-BUTTONS");
```

In Release 12 and earlier versions of AutoCAD, you could assign any kind of menu to any other. For example, you could assign a SCREEN menu to a POP menu. With Release 13 and later versions of AutoCAD, you can assign menus to other menus on the Windows platform only if they are of the same type. A POP menu can be assigned only to another POP menu, and a SCREEN menu to another SCREEN menu. You can specify the menu in detail, because Windows loads partial menus.

Calling `acedMenuCmd()` and passing "P1=test.numeric" assigns POP menu 12 to POP menu 2, assuming that the following menu file definitions exist.

```
***MENUGROUP=test
***POP12
**NUMERIC
[Numeric Menu]
[First item]
[Second item]
```

The following call shows how to activate a drop-down menu and then display it.

```
acedMenuCmd("P1=NUMERIC");
```

The call to `acedMenuCmd()` assigns the submenu NUMERIC to drop-down menu 1 (in the upper-left corner of the graphics screen).

See the *AutoCAD Customization Guide* for more information on custom menus.

Control of Graphics and Text Screens

On single-screen AutoCAD installations, an ObjectARX application can call `acedGraphScr()` to display the graphics screen or `acedTextScr()` to display the text screen. These functions are equivalent to the AutoCAD GRAPHSCR and TEXTSCR commands or to toggling the Flip Screen function key. The function `acedTextPage()` is like `acedTextScr()`, but it clears the text screen before displaying it (as the AutoCAD STATUS command does).

The `acedRedraw()` function is similar to the AutoCAD REDRAW command, but it provides more control over what is displayed: it can redraw the entire graphics screen and also specify a single object to be either redrawn or undrawn (blanked out). If the object is a complex object such as a polyline or block, `acedRedraw()` can draw (or undraw) either the entire object or only its header. The `acedRedraw()` function can be used also to highlight or unhighlight selected objects.

Control of Low-Level Graphics and User Input

Certain functions provide direct access to the AutoCAD graphics screen and input devices. They enable ObjectARX applications to use some of the display and user-interaction facilities built into AutoCAD.

The `acedGrText()` function displays text in the status or menu areas, with or without highlighting. The `acedGrDraw()` function draws a vector in the current viewport, with control over color and highlighting. The `acedGrVecs()` function draws multiple vectors. The `acedGrRead()` function returns "raw" user input, whether from the keyboard or the pointing device; if the call to `acedGrRead()` enables tracking, the function returns digitized coordinates that can be used for dragging.

WARNING! Because these functions depend on code in AutoCAD, their operation can change from release to release. Applications that call these functions may not be upward compatible. Also, they depend on the current hardware configuration. In particular, applications that call `acedGrText()` and `acedGrRead()` are not likely to work the same on all configurations unless the developer uses them as described earlier to avoid hardware-specific features. These functions do almost no error reporting and can damage the graphics screen display (see the example for a way to fix this problem).

The following sequence reverses damage to the graphics screen display caused by incorrect calls to `acedGrText()`, `acedGrDraw()`, or `acedGrVecs()`.

```
acedGrText(-3, NULL, 0);
acedRedraw(NULL, 0);
```

The arguments to `acedGrText()` have the following meanings: -3 restores standard text, `NULL` == no new text, and 0 == no highlighting. The arguments to `acedRedraw()` have the following meanings: `NULL` == all entities, and 0 == entire viewport.

Tablet Calibration

AutoCAD users with a digitizing tablet can calibrate the tablet by using the TABLET command. With the `acedTablet()` function, applications can manage calibrations by setting them directly and by saving calibration settings for future use. The function takes two arguments, `list` and `result`, each of which is a result-buffer list. The first result buffer in the first list is an integer code that must be 0 to retrieve the current calibration (in `result`), or 1 to set the calibration according to the remaining buffers in list. Calibrations are expressed as four 3D points (in addition to the code). The first three of these points—row1, row2, and row3—are the three rows of the tablet's transformation matrix. The fourth point is a vector, direction, that is normal to the plane of the tablet's surface (expressed in WCS).

NOTE The TABMODE system variable controls whether Tablet mode is set to On (1) or Off (0). You can control it by using `acedSetVar()`.

The following code sequence retrieves the current tablet calibration, and saves it in `calibr2`. In this example, the user has used the TABLET command to calibrate the matrix, and Tablet mode is on.

```
struct resbuf *calibr1, *calibr2;
struct resbuf varbuf, rb;

// Retrieve the current calibration.
calibr1 = acutBuildList(RTSHORT, 0, RTNONE);

if (acedTablet(calibr1, &calibr2) != RTNORM) {
    acdbFail("Calibration not obtainable\n");
    return BAD;
}
```

The code returned in the result argument, `calibr2` in the example, is automatically set to 1. To reset the calibration to the values retrieved by the preceding example, you could use the following code:

```
if (acedTablet(calibr2, &calibr1) != RTNORM) {
    acdbFail("Couldn't reset calibration\n");
    return BAD;
}
rb.restype = RTSHORT;
rb.resval.rint = 1;
acedSetVar("TABMODE", &rb);
acedGetVar("TABMODE" &varbuf);
if (varbuf.resval.rint == 0) {
    acdbFail("Couldn't set TABMODE\n");
    return BAD;
}
```

In this example, `calibr1` now contains the result of the calibration. Because this is presumably identical to `calibr2` (which was initialized by `acedTablet()`), you don't necessarily need this result. When you set a calibration, you can specify a `NULL` result, which causes `acedTablet()` to set the calibration "silently."

```
if (acedTablet(calibr2, NULL) != RTNORM) { . . . }
```

The transformation matrix passed as row1, row2, and row3 is a 3x3 transformation matrix 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, the scale factor D', yielding the point $(X'/D', Y'/D')$.

For a projective transformation, which is the most general case, `acedTablet()` 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') .

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} 0, but $M_{00} = M_{11}$ and $M_{10} = {}^{-}M_{01}$.

NOTE When you set a calibration, the result does not equal the list argument if the direction in the list was not normalized; AutoCAD normalizes the direction vector before it returns it. Also, it ensures that the third element in the third column (row3[Z]) is equal to 1. This situation should not arise if you set the calibration using values retrieved from AutoCAD by means of `acedTablet()`. However, it can happen if your program calculates the transformation itself.

Wild-Card Matching

The `acutWcMatch()` function enables applications to compare a string to a wild-card pattern. This facility can be used when building a selection set (in conjunction with `acedSSGet()`) and when retrieving extended entity data by application name (in conjunction with `acdbEntGetX()`).

The `acutWcMatch()` function compares a single string to a pattern, and returns `RTNORM` if the string matches the pattern, and `RTERROR` 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, and an asterisk (*) matches a sequence of characters; certain other special characters have meanings within the pattern. For a complete table of characters used in wild-card strings, see the description of `acutWcMatch()`.

In the following examples, a string variable called `matchme` has been declared and initialized. The following call checks whether `matchme` begins with the five characters “allof”.

```
if (acutWcMatch(matchme, "allof*") == RTNORM) {
    .
    .
    .
}
```

The following call illustrates the use of brackets in the pattern. In this case, `acutWcMatch()` returns `RTNORM` if `matchme` equals “STR1”, “STR2”, “STR3”, or “STR8”.

```
if (acutWcMatch(matchme, "STR[1-38]") == RTNORM) {
    .
    .
    .
}
```

The pattern string can specify multiple patterns, separated by commas. The following call returns `RTNORM` if `matchme` equals “ABC”, if it begins with “XYZ”, or if it ends with “123”.

```
if (acutWcMatch(matchme, "ABC,XYZ*,*123") == RTNORM) {
    .
    .
    .
}
```

The `acutWcMatchEx()` function is similar to `acutWcMatch()`, but it has an additional argument to allow it to ignore case.

```
bool
acutWcMatchEx(
    const char * string,
    const char * pattern,
    bool ignoreCase);
```

Part III

Defining New Classes

Deriving a Custom ObjectARX Class

11

In This Chapter

- Custom Class Derivation
- Runtime Class Identification
- Class Declaration Macro
- Class Implementation Macros
- Class Initialization Function

This chapter describes how to use the ObjectARX macros to simplify the task of deriving a custom ObjectARX class. These macros allow a custom class to participate in the `AcRxObject` runtime type identification mechanism. If you do not need to distinguish your custom class at runtime, you can use standard C++ derivation style to create the new class.

Custom Class Derivation

ObjectARX provides a set of macros, declared in the *rxboiler.h* file, that helps you create new classes derived from `AcRxObject`. You can derive new classes from most of the classes in the ObjectARX hierarchy except the AutoCAD Release 12 entity set (listed in chapter 6, "Entities,") and the symbol table classes. If you do not use the ObjectARX macros to define your new class, the class will inherit the runtime identity of its most immediate ObjectARX-registered parent class.

Applications can most efficiently derive new classes from the following classes:

- `AcRxObject`
- `AcRxService`
- `AcDbObject`
- `AcDbEntity`
- `AcDbCurve`
- `AcDbObjectReactor`
- `AcDbDatabaseReactor`
- `AcDbEntityReactor`
- `AcTransactionReactor`
- `AcEdJig`
- `AcEditorReactor`

Applications should not derive classes from the following:

- `AcDbAttribute`
- `AcDbAttributeDefinition`
- `AcDbArc`
- `AcDbBlockReference`
- `AcDbCircle`
- `AcDbFace`
- `AcDbLine`
- `AcDbMInsertBlock`
- `AcDbPoint`
- `AcDbShape`
- `AcDbSolid`
- `AcDbText`
- `AcDbTrace`
- All `AcDbXxxDimension` classes
- `AcDbViewport`
- `AcDbGroup`
- All classes derived from `AcDbSymbolTable`
- All classes derived from `AcDbSymbolTableRecord`
- `AcDbBlockBegin`
- `AcDbBlockEnd`

- AcDbSequenceEnd
- AcDb2dPolyline
- AcDb2dPolylineVertex
- AcDb3dPolyline
- AcDb3dPolylineVertex
- AcDbPolygonMesh
- AcDbPolygonMeshVertex
- AcDbPolyFaceMesh
- AcDbPolyFaceMeshVertex
- AcDbFaceRecord

Classes not appearing in either of the preceding lists can theoretically be derived from, although doing so is not explicitly supported.

Runtime Class Identification

Every class in the ObjectARX hierarchy that is derived from `AcRxObject` has a corresponding class descriptor object, which is an instance of `AcRxClass` that holds information for runtime type identification. The class descriptor object, `gpDesc`, is a static data member of the class—for example, `AcDbEllipse::gpDesc`. Class descriptor objects are created at initialization, when classes are registered with ObjectARX and are added to a system-level dictionary, `acrxClassDictionary`. The macros described here facilitate the declaration and implementation of certain functions related to runtime identification and initialization functions. These include the class initialization routine as well as the `desc()`, `cast()`, `isKindOf()`, and `isA()` functions for the custom class.

Important functions provided by the `AcRxObject` class for runtime type identification include the following:

- `desc()`, a static member function that returns the class descriptor object of a particular (known) class.
- `cast()`, a static member function that returns an object of the specified type, or `NULL` if the object is not of the required class (or a derived class).
- `isKindOf()` returns whether an object belongs to the specified class (or a derived class).
- `isA()` returns the class descriptor object of an object whose class is unknown.

When you want to know what class an object is, use `AcRxObject::isA()`. This function returns the class descriptor object (an instance of `AcRxClass`) for a database object. Its signature is

```
AcRxClass* isA() const;
```

When you already know what class an object is, you can use the `desc()` function to obtain the class descriptor object:

```
static AcRxClass* desc();
```

The following example looks for instances of `AcDbEllipse` or any class derived from it, using `isKindOf()` and the `AcDbEllipse::desc()` static member function:

```
AcDbEntity* curEntity = somehowGetAndOpenAnEntity();

if (curEntity->isKindOf(AcDbEllipse::desc())) {
   // Got some kind of AcDbEllipse instance.
}
```

This example shows another way of looking for instances of `AcDbEllipse`, or any class derived from it, using the `AcDbEllipse::cast()` static member function:

```
AcDbEllipse* ellipseEntity = AcDbEllipse::cast(curEntity);

if (ellipseEntity != NULL) {
   // Got some kind of AcDbEllipse instance.
}
```

The following example looks for instances of `AcDbEllipse`, but not instances of classes derived from `AcDbEllipse`, using `isA()` and `AcDbEllipse::desc()`:

```
if (curEntity->isA() == AcDbEllipse::desc()) {
   // Got an AcDbEllipse, no more, no less.
```

Class Declaration Macro

The header file for a custom class can use the `ACRX_DECLARE_MEMBERS(CLASS_NAME)` ObjectARX macro to declare the `desc()`, `cast()`, and `isA()` functions.

This macro is used in the public section of the class declaration, as follows:

```
class myClass : public AcRxObject
{
public:
ACRX_DECLARE_MEMBERS(myClass);
...
};
```

For `AsdkPoly`, the following line expands to a single long line of code.

```
ACRX_DECLARE_MEMBERS(AsdkPoly);
```

When reformatted to multiple lines for clarity, the line looks like this:

```
virtual AcRxClass* isA() const;
static AcRxClass* gpDesc;
static AcRxClass* desc();
static AsdkPoly* cast(const AcRxObject* inPtr)
{
    return ((inPtr == 0)
        || !inPtr->isKindOf(AsdkPoly::desc()))
        ? 0 : (AsdkPoly*)inPtr;
};
static void rxInit();
```

The static `rxInit()` function and the static `gpDesc` pointer declared by this macro are used to implement the `isA()`, `desc()`, and `cast()` functions.

Class Implementation Macros

To implement your custom class, use one of these three macros in the source file:

- `ACRX_NO_CONS_DEFINE_MEMBERS(CLASS_NAME, PARENT_CLASS)`

  Use for abstract classes and any other classes that should not be instantiated.
- `ACRX_CONS_DEFINE_MEMBERS(CLASS_NAME, PARENT_CLASS, VERNO)`

  Use for transient classes that can be instantiated but are not written to file.
- `ACRX_DXF_DEFINE_MEMBERS(CLASS_NAME, PARENT_CLASS, DWG_VERSION,\ MAINTENANCE_VERSION, PROXY_FLAGS, DXF_NAME, APP)`

  Use for classes that can be written to, or read from, DWG and DXF files. Each of these macros defines the following:

  - Class descriptor object
  - Class initialization function (see "Class Initialization Function" on page 289)
  - A `desc()` function for this class
  - A virtual `isA()` function (inherited from `AcRxObject`) that this custom class will override

For `AsdkPoly`, the following line expands to a very long single line of code:

```
ACRX_DXF_DEFINE_MEMBERS(AsdkPoly, AcDbCurve, AcDb::kDHL_CURRENT,\
    AcDb::kMReleaseCurrent, 0, POLYGON, /*MSG0*/"AutoCAD");
```

When reformatted to multiple lines for clarity, the line looks like this:

```
AcRxClass* AsdkPoly::desc()
{
    if (AsdkPoly::gpDesc != 0)
        return AsdkPoly::gpDesc;
    return AsdkPoly::gpDesc =
        (AcRxClass*)((AcRxDictionary*)acrxSysRegistry()->
        at("ClassDictionary"))->at("AsdkPoly");
}
AcRxClass* AsdkPoly::isA() const
{
    return AsdkPoly::desc();
}
AcRxClass* AsdkPoly::gpDesc = 0;
static AcRxObject * makeAsdkPoly()
{
    return new AsdkPoly();
}
void AsdkPoly::rxInit()
{
    if (AsdkPoly::gpDesc != 0)
        return;
    AsdkPoly::gpDesc = newAcRxClass("AsdkPoly",
        "AsdkCurve", AcDb::kDHL_CURRENT, AcDb::kMReleaseCurrent,
         0, &makeAsdkPoly, "POLYGON", "\"AutoCAD\"");
};
```

When expanded, the semicolon (;) at the end of the macro call line moves to just after the closing brace (}) for a function definition. Therefore, this semicolon is not required for this macro call line.

If you want to write your own `rxInit()` function, use the `ACRX_DEFINE_MEMBERS()` macro by itself, which defines `desc()`, `cast()`, and `isA()` for your class but does not define the `rxInit()` function. This macro also does not create the associated `AcRxClass` object, which is the responsibility of the `rxInit()` function.

Class Initialization Function

The class initialization function for each class is `rxInit()`. An application that defines a custom class must invoke this function during runtime initialization.

This function is defined automatically by each of the three `ACRX_xxx_DEFINE_MEMBERS()` macros and performs the following tasks:

- Registers the custom class
- Creates the class descriptor object
- Places the class descriptor object in the class dictionary

If you want to define your own `rxInit()` function, use the `ACRX_DEFINE_MEMBERS()` macro.

Deriving from AcDbObject

12

In This Chapter

- Overriding AcDbObject Virtual Functions
- Implementing Member Functions
- Filing Objects to DWG and DXF Files
- Object References
- Ownership References
- Pointer References
- Long Transaction Issues for Custom Objects
- Purge
- Undo and Redo
- subErase, subOpen, subClose, and subCancel
- Example of a Custom Object Class
- Object Version Support

This chapter describes how to derive a custom class from `AcDbObject`. It provides detailed information on filers, the four types of object references (hard and soft owners, and hard and soft pointers), and the undo and redo operations. This chapter also discusses mechanisms for object versioning.

The descriptions in this chapter assume you are familiar with the material described in chapter 5, "Database Objects," and chapter 11, "Deriving a Custom ObjectARX Class."

Overriding AcDbObject Virtual Functions

If you're subclassing from `AcDbObject`, there are a number of virtual functions you must override, as shown in the following sections. These sections show which other functions are usually overridden and which functions are only rarely overridden.

AcDbObject: Essential Functions to Override

A custom class must override the following functions:

```
virtual Acad::ErrorStatus
dwgInFields(AcDbDwgFiler* filer);

virtual Acad::ErrorStatus
dwgOutFields(AcDbDwgFiler* filer) const;

virtual Acad::ErrorStatus
dxfInFields(AcDbDxfFiler* filer);

virtual Acad::ErrorStatus
dxfOutFields(AcDbDxfFiler* filer) const;

<Destructor>
```

AcDbObject: Functions Often Overridden

A custom class often overrides the following functions:

```
virtual Acad::ErrorStatus
audit(AcDbAuditInfo*);

// Commonly useful, as this happens at a point where a new
// object state is being committed.
//
virtual Acad::ErrorStatus subClose();

// The next two functions apply to container objects.
//
virtual Acad::ErrorStatus
deepClone(AcDbObject* pOwnerObject,
          AcDbObject*& pClonedObject,
          AcDbIdMapping& idMap,
          Adesk::Boolean isPrimary = Adesk::kTrue) const;

virtual Acad::ErrorStatus
wblockClone(AcRxObject* pOwnerObject,
            AcDbObject*& pClonedObject,
            AcDbIdMapping& idMap,
            Adesk::Boolean isPrimary = Adesk::kTrue) const;
```

AcDbObject: Functions Sometimes Overridden

A custom class sometimes overrides the following functions:

```
virtual Acad::ErrorStatus
subErase(Adesk::Boolean erasing);

virtual Acad::ErrorStatus
subHandOverTo(AcDbObject* newObject);

virtual Acad::ErrorStatus
subOpen(AcDb::OpenMode);

virtual Acad::ErrorStatus
subCancel();

virtual Acad::ErrorStatus
subSwapIdWith(AcDbObjectId otherId,
              Adesk::Boolean swapXdata = Adesk::kFalse);
```

AcDbObject: Functions Rarely Overridden

A custom class rarely overrides the following functions:

```
virtual Acad::ErrorStatus
setOwnerId(AcDbObjectId);

virtual resbuf*
xData(const char* regappName = NULL) const;

virtual Acad::ErrorStatus
setXData(const resbuf* xdata);

virtual void
addPersistentReactor(AcDbObjectId objId);

virtual Acad::ErrorStatus
removePersistentReactor(AcDbObjectId objId);

virtual void
cancelled(const AcDbObject* dbObj);

virtual void
copied(const AcDbObject* dbObj,
       const AcDbObject* newObj);

virtual void
erased(const AcDbObject* dbObj,
       Adesk::Boolean pErasing = Adesk::kTrue);

virtual void
goodbye(const AcDbObject* dbObj);
```

```
virtual void
openedForModify(const AcDbObject* dbObj);

virtual void
modified(const AcDbObject* dbObj);

virtual void
modifyUndone(const AcDbObject* dbObj);

virtual void
modifiedXData(const AcDbObject* dbObj);

virtual void
unappended(const AcDbObject* dbObj);

virtual void
objectClosed(const AcDbObjectId objId);

virtual void
modifiedGraphics(const AcDbEntity* dbEnt);
```

AcRxObject: Functions Rarely Overridden

A custom class rarely overrides these functions:

```
virtual AcRxObject*
clone() const;

virtual void
copyFrom(const AcRxObject* pSrc);

// Do not override; AcDbObject behavior is already accounted for.
//

virtual HRESULT __stdcall
QueryInterface ( REFIID riid,
                 void ** ppvObject );

virtual ULONG __stdcall
AddRef();

virtual ULONG __stdcall
Release();
```

AcDbEntity: Functions to Override

If you are implementing a custom entity, see chapter 13, "Deriving from AcDbEntity," for a list of the functions to override.

AcDbCurve: Functions to Override

A custom class must override the following functions:

```
virtual Adesk::Boolean
isClosed() const;

virtual Adesk::Boolean
isPeriodic() const;

virtual Adesk::Boolean
isPlanar() const;

virtual Acad::ErrorStatus
getPlane(AcGePlane&, AcDb::Planarity&) const;

virtual Acad::ErrorStatus
getStartParam(double&) const;

virtual Acad::ErrorStatus
getEndParam(double&) const;

virtual Acad::ErrorStatus
getStartPoint(AcGePoint3d&) const;

virtual Acad::ErrorStatus
getEndPoint(AcGePoint3d&) const;

virtual Acad::ErrorStatus
getPointAtParam(double, AcGePoint3d&) const;

virtual Acad::ErrorStatus
getParamAtPoint(const AcGePoint3d&, double&)const;

virtual Acad::ErrorStatus
getDistAtParam(double param, double& dist) const;

virtual Acad::ErrorStatus
getParamAtDist(double dist, double& param) const;

virtual Acad::ErrorStatus
getDistAtPoint(const AcGePoint3d&, double&) const;

virtual Acad::ErrorStatus
getPointAtDist(double, AcGePoint3d&) const;

virtual Acad::ErrorStatus
getFirstDeriv(
    double param,
    AcGeVector3d& firstDeriv) const;
virtual Acad::ErrorStatus
getFirstDeriv(
    const AcGePoint3d&,
    AcGeVector3d& firstDeriv) const;
```

```
virtual Acad::ErrorStatus
getSecondDeriv(
    double param,
    AcGeVector3d& secDeriv) const;

virtual Acad::ErrorStatus
getSecondDeriv(
    const AcGePoint3d&,
    AcGeVector3d& secDeriv) const;

virtual Acad::ErrorStatus
getClosestPointTo(
    const AcGePoint3d& givenPnt,
    AcGePoint3d& pointOnCurve,
    Adesk::Boolean extend
    = Adesk::kFalse) const;

virtual Acad::ErrorStatus
getClosestPointTo(
    const AcGePoint3d& givenPnt,
    const AcGeVector3d& normal,
    AcGePoint3d& pointOnCurve,
    Adesk::Boolean extend
    = Adesk::kFalse) const;

virtual Acad::ErrorStatus
getOrthoProjectedCurve(
    const AcGePlane&,
    AcDbCurve*& projCrv) const;

virtual Acad::ErrorStatus
getProjectedCurve(
    const AcGePlane&,
    const AcGeVector3d& projDir,
    AcDbCurve*& projCrv) const;

virtual Acad::ErrorStatus
getOffsetCurves(
    double offsetDist,
    AcDbVoidPtrArray& offsetCurves) const;

virtual Acad::ErrorStatus
getSpline(AcDbSpline*& spline) const;

virtual Acad::ErrorStatus
getSplitCurves(
    const AcGeDoubleArray& params,
    AcDbVoidPtrArray& curveSegments) const;

virtual Acad::ErrorStatus
getSplitCurves(
    const AcGePoint3dArray& points,
    AcDbVoidPtrArray& curveSegments) const;

virtual Acad::ErrorStatus
extend(double newParam);
```

```
virtual Acad::ErrorStatus
extend(
    Adesk::Boolean extendStart,
    const AcGePoint3d& toPoint);

virtual Acad::ErrorStatus
getArea(double&) const;
```

Implementing Member Functions

When you define a new member function or override an existing function, the first call you usually make is `assertReadEnabled()`, `assertWriteEnabled()`, or `assertNotifyEnabled()` to verify that the object is open in the correct state. Of these three functions, `assertWriteEnabled()` is the most important. You can use this function to control undo recording of the modification that is occurring in the member function. (See "Undo and Redo" on page 324.) Even if you don't desire undo recording, it is essential to call

```
assertWriteEnabled(kFalse, kFalse);
```

This call marks the object for incremental save. Failure to follow this instruction can result in corrupted drawings.

The following table shows the three possible states for opening an object (read, write, notify) and indicates which assert calls succeed for each state. If the object is not open in one of the allowed states for the assert function call, the function does not return. AutoCAD exits, and the user is prompted to save the drawing.

	Object open for		
	Read	*Write*	*Notify*
assertReadEnabled()	returns	returns	returns
assertWriteEnabled()	aborts	returns	aborts
assertNotifyEnabled()	returns	returns	returns

Filing Objects to DWG and DXF Files

When deriving a class from `AcDbObject`, you need the additional information on the AutoCAD filing mechanism provided in this chapter. The following four functions are used for filing objects to DWG and DXF files. They are also used for other purposes, such as cloning.

```
Acad::ErrorStatus
AcDbObject::dwgOut(AcDbDwgFiler* filer);

Acad::ErrorStatus
AcDbObject::dwgIn(AcDbDwgFiler* filer);

Acad::ErrorStatus
AcDbObject::dxfOut(
    AcDbDxfFiler* filer,
    Adesk::Boolean allXdFlag,
    Adesk::uchar* regAppTable) const);

Acad::ErrorStatus
AcDbObject::dxfIn(AcDbDxfFiler* filer);
```

Each function takes a pointer to a filer as its primary argument. An `AcDbObject` writes data to and reads data from a filer. The `FilerType` enum allows you to check the filer type. Filer types are

- `kFileFiler` (used for DWG and DXF files)
- `kCopyFiler`
- `kUndoFiler`
- `kBagFiler` (used with `acdbEntMake()`, `acdbEntMod()`, and `acdbEntGet()`)
- `kIdXlateFiler`
- `kPageFiler`
- `kDeepCloneFiler`
- `kWBlockCloneFiler`
- `kPurgeFiler`

The `dwgOut()` and `dwgIn()` functions in turn call `dwgOutFields()` and `dwgInFields()`, respectively, and the DXF filing functions call an analogous set of functions for DXF. If you are deriving a custom class from `AcDbObject`, you will need to override the following virtual functions, which are used for persistent storage of objects as well as for copying and undo operations:

- `dwgOutFields()`
- `dwgInFields()`
- `dxfOutFields()`
- `dxfInFields()`

dwgOut() Function

The `dwgOut()` function, which calls `dwgOutFields()`, is invoked by the following commands and conditions:

- SAVE (uses `kFileFiler`)
- SAVEAS (uses `kFileFiler`)
- WBLOCK (uses `kWblockCloneFiler` and `kIdXlateFiler`)
- INSERT, XREF (use `kDeepCloneFiler` and `kIdXlateFiler`)
- COPY (uses same filers as INSERT; a copy requires writing out an object's state and then reading it back in to an object of the same class)
- PURGE (uses a `kPurgeFiler`)
- Any time an object is paged out (uses a `kPageFiler`)
- Any time an object is modified (for undo recording; uses a `kUndoFiler`)

dwgIn() Function

The `dwgIn()` function, which calls `dwgInFields()`, is invoked by the following commands and conditions:

- OPEN (uses a `kFileFiler`)
- UNDO (uses a `kUndoFiler`)
- INSERT, COPY, XREF (use a `kDeepCloneFiler` and a `kIdXlateFiler`)
- WBLOCK (uses `kWblockCloneFiler` and `kIdXlateFiler`)
- Any time an object is paged in (uses a `kPageFiler`)

dxfOut() Function

The `dxfOut()` function, which calls `dxfOutFields()`, is invoked by the following commands and functions:

- WBLOCK
- SAVE
- SAVEAS
- `acdbEntGet()`

dxfIn() Function

The `dxfIn()` function, which calls `dxfInFields()`, is invoked by the following commands and functions:

- OPEN
- INSERT
- `acdbEntMod()` or `acdbEntMake()`

Error Checking

When you are writing to a filer, you do not need to perform intermediate error checking. Once an error condition is encountered, the filer returns the same error status to all write requests until the status is cleared by the filer.

Every filer class has a `getFilerStatus()` function that returns the filer status. When you are reading in a file, you may want to check the filer status if you rely on success or failure for your next step.

Implementing the DWG Filing Functions

If you are implementing `dwgOutFields()` and `dwgInFields()` for a new class, you must first call `assertReadEnabled()` or `assertWriteEnabled()` to ensure that the object is open in the correct state.

The next thing your derived class must do is to call the same function (for example, `dwgOutFields()`) on the parent class. This process is referred to as super messaging. The following is an example:

```
AcDbDerivedClass::dwgOutFields( ... );
{
   assertReadEnabled()
   myParent::dwgOutFields();

   // Perform class-specific operations after super-messaging.
}
```

If you forget to call the corresponding message of the parent class, you'll receive a runtime error.

After super-messaging, you write or read fields. You may improve performance by checking the filer type. For example, if the filer type is `kIdXlateFiler` and your class doesn't define any reference connections, you can simply return.

With DWG files, you need to write and read calls in the same order. If calls are mismatched, derived classes will be confused. If you have any variable-sized data, put the count first.

Sample Code for dwgOutFields()

Most of the filer calls are `writeItem()`, a member function that has been overloaded for all supported data types. There are also other functions, such as `writeInt32()` used in the following example, that can be used to support automatic type casting. Such functions force the argument to be treated as the specified type regardless of its actual type in memory.

NOTE If your class has integer data members, you need to use the read and write functions that explicitly state the integer size (for example, `writeInt32`).

The following is sample code from `AsdkPoly::dwgOutFields()`:

```
Acad::ErrorStatus
AsdkPoly::dwgOutFields(AcDbDwgFiler* filer) const
{
    assertReadEnabled();
    Acad::ErrorStatus es;
    if ((es = AcDbCurve::dwgOutFields(filer))
        != Acad::eOk)
    {
        return es;
    }

    // Object Version - must always be the first item.
    //
    Adesk::Int16 version = VERSION;
    filer->writeItem(version);

    filer->writePoint2d(mCenter);
    filer->writePoint2d(mStartPoint);
    filer->writeInt32(mNumSides);
    filer->writeVector3d(mPlaneNormal);
    filer->writeString(mpName);
    // mTextStyle is a hard pointer id, so filing it out to
    // the purge filer (kPurgeFiler) prevents purging of
    // this object.
    //
    filer->writeHardPointerId(mTextStyle);
    filer->writeDouble(mElevation);
    return filer->filerStatus();
}
```

Sample Code for dwgInFields()

The following is sample code for `AsdkPoly::dwgInFields()`:

```
Acad::ErrorStatus
AsdkPoly::dwgInFields(AcDbDwgFiler* filer)
{
    assertWriteEnabled();
    Acad::ErrorStatus es;
    if ((es = AcDbCurve::dwgInFields(filer)) != Acad::eOk)
    {
        return es;
    }

    // Object Version - must always be the first item.
    //
```

```
    Adesk::Int16 version;
    filer->readItem(&version);
    if (version > VERSION)
        return Acad::eMakeMeProxy;
    switch (version)
    {
    case 1:
    {
        AcGePoint3d center;
        filer->readPoint3d(&center);
        AcGePoint3d startPoint;
        filer->readPoint3d(&startPoint);
        filer->readInt32(&mNumSides);
        filer->readVector3d(&mPlaneNormal);
        acutDelString(mpName);
        filer->readString(&mpName);
        filer->readHardPointerId(&mTextStyle);

        //convert data from old format
        acdbWcs2Ecs(asDblArray(center),asDblArray(center),
            asDblArray(mPlaneNormal),Adesk::kFalse);
        mCenter.set(center.x,center.y);
        mElevation = center.z;

        acdbWcs2Ecs(asDblArray(startPoint),asDblArray(startPoint),
            asDblArray(mPlaneNormal),Adesk::kFalse);
        mStartPoint.set(startPoint.x,startPoint.y);
        assert(mElevation == startPoint.z);
        break;
    }
    case 2:
        filer->readPoint2d(&mCenter);
        filer->readPoint2d(&mStartPoint);
        filer->readInt32(&mNumSides);
        filer->readVector3d(&mPlaneNormal);
        acutDelString(mpName);
        filer->readString(&mpName);
        filer->readHardPointerId(&mTextStyle);
        filer->readDouble(&mElevation);
        break;
    default:
        assert(false);
    }
    return filer->filerStatus();
}
```

Implementing the DXF Filing Functions

If you are implementing `dxfOutFields()` and `dxfInFields()` for a new class, your derived class must first call `assertReadEnabled()` or `assertWriteEnabled()`. It must then call the same function on the parent class (super-messaging).

DXF Group Code Ranges

The DXF representation of an object is composed of pairs of group codes and data, with each group code mapping to a specific data type. When you define your own DXF representation, the first data group you write out and read in must be a subclass data marker. This marker consists of a 100 group code followed by a string that is the current class name. Then, you select group codes from the following table that correspond to the data types of each data field you are writing out.

DXF group code ranges for object representation

From	To	Data Type
1	4	Text
6	9	Text
10	17	Point or vector (3 reals)
38	59	Real
60	79	16-bit integer
90	99	32-bit integer
100	100	Subclass data marker
102	102	Text
140	149	Real
170	179	16-bit integer
210	219	3 reals
270	279	16-bit integer
280	289	8-bit integer
300	309	Text
310	319	Binary chunk
320	329	Handle
330	339	Soft pointer ID
340	349	Hard pointer ID

DXF group code ranges for object representation (*continued*)		
From	**To**	**Data Type**
350	359	Soft owner ID
360	369	Hard owner ID

An object ID translates to an `rlname`. For example, an `AcDbObjectId` corresponds to an `ads_name`, which is represented in the `resval` union as `rlname`.

Order Dependence

With DXF, at the class author's discretion, data groups can be presented in arbitrary order, or optionally omitted. Some classes support order independence of data groups, while others do not. If you allow order independence, then your `dxfInFields()` function must use a switch statement to choose an action based on the group code value. Order independence is usually appropriate for objects with a fixed and predictable set of fields. Objects with variable-length arrays or structures tend to be order-dependent when they are filed out and in.

Sample Code for dxfOutFields()

The following is sample code from `AsdkPoly::dxfOutFields()`:

```
Acad::ErrorStatus
AsdkPoly::dxfOutFields(AcDbDxfFiler* filer) const
{
    assertReadEnabled();
    Acad::ErrorStatus es;
    if ((es = AcDbCurve::dxfOutFields(filer))
        != Acad::eOk)
    {
        return es;
    }
    filer->writeItem(AcDb::kDxfSubclass, "AsdkPoly");
    // Object Version
    //
    Adesk::Int16 version = VERSION;
    filer->writeInt16(AcDb::kDxfInt16, version);
    filer->writePoint2d(AcDb::kDxfXCoord, mCenter);
    filer->writePoint2d(AcDb::kDxfXCoord + 1, mStartPoint);
    filer->writeInt32(AcDb::kDxfInt32, mNumSides);

    // Always use max precision when writing out the normal.
    filer->writeVector3d(AcDb::kDxfNormalX, mPlaneNormal,16);
    filer->writeString(AcDb::kDxfText, mpName);
    filer->writeItem(AcDb::kDxfHardPointerId, mTextStyle);
    filer->writeDouble(AcDb::kDxfReal, mElevation);
    return filer->filerStatus();
}
```

Sample Code for dxfInFields() with Order Independence

The following is sample code for `AsdkPoly::dxfInFields()`:

```
Acad::ErrorStatus
AsdkPoly::dxfInFields(AcDbDxfFiler* filer)
{
    assertWriteEnabled();
    Acad::ErrorStatus es = Acad::eOk;
    resbuf rb;
    if ((AcDbCurve::dxfInFields(filer) != Acad::eOk)
        || !filer->atSubclassData("AsdkPoly"))
    {
        return filer->filerStatus();
    }
    // Object Version
    Adesk::Int16 version;
    filer->readItem(&rb);
    if (rb.restype != AcDb::kDxfInt16)
    {
        filer->pushBackItem();
        filer->setError(Acad::eInvalidDxfCode,
            "\nError: expected group code %d (version)",
            AcDb::kDxfInt16);
        return filer->filerStatus();
    }

    version = rb.resval.rint;
    if (version > VERSION)
        return Acad::eMakeMeProxy;

    AcGePoint3d cen3d,sp3d;
    AcGePoint2d cen2d,sp2d;
    long numSides;
    AcDbObjectId textStyle;
    double elevation;
    Adesk::UInt32 fieldsFlags = 0;
    char * pName = NULL;
    AcGeVector3d planeNormal;

    while ((es == Acad::eOk)
        && ((es = filer->readResBuf(&rb)) == Acad::eOk))
    {
        switch (rb.restype) {
        case AcDb::kDxfXCoord:
            if (version == 1)
                cen3d = asPnt3d(rb.resval.rpoint);
            else
                cen2d = asPnt2d(rb.resval.rpoint);
            fieldsFlags |= 0x1;
            break;
```

```
        case AcDb::kDxfXCoord + 1:
            if (version == 1)
                sp3d = asPnt3d(rb.resval.rpoint);
            else
                sp2d = asPnt2d(rb.resval.rpoint);
            fieldsFlags |= 0x2;
            break;
        case AcDb::kDxfInt32:
            numSides = rb.resval.rlong;
            fieldsFlags |= 0x4;
            break;
        case AcDb::kDxfNormalX:
            planeNormal = asVec3d(rb.resval.rpoint);
            fieldsFlags |= 0x8;
            break;
        case AcDb::kDxfText:
            acutUpdString(rb.resval.rstring,pName);
            fieldsFlags |= 0x11;
            break;
        case AcDb::kDxfHardPointerId:
            acdbGetObjectId(textStyle, rb.resval.rlname);
            fieldsFlags |= 0x12;
            break;
        case AcDb::kDxfReal:
            if (version == 2)
            {
                fieldsFlags |= 0x10;
                elevation = rb.resval.rreal;
                break;
            }
            //fall through intentional
        default:
            // An unrecognized group. Push it back so that
            // the subclass can read it again.
            filer->pushBackItem();
            es = Acad::eEndOfFile;
            break;
        }
    }

    // At this point, the es variable must contain eEndOfFile,
    // either from readResBuf() or from pushbackBackItem(). If
    // not, it indicates that an error happened and we should
    // return immediately.
    //
    if (es != Acad::eEndOfFile)
        return Acad::eInvalidResBuf;
    // Now check to be sure all necessary group codes were
    // present.
    //
    // Mandatory fields:
    // - center
    // - start point
    // - normal
    // - number of sides
    // - elevation (if version > 1)
```

```
    short required[]  =
        {AcDb::kDxfXCoord, AcDb::kDxfXCoord+1, AcDb::kDxfInt32,
        AcDb::kDxfNormalX, AcDb::kDxfReal};

    for (short i = 0; i < (version>1?4:3); i++) {
        if (!fieldsFlags & 0x1) {
            filer->setError(Acad::eMissingDxfField,
                "\nMissing DXF group code: %d", 2, required[i]);
            return Acad::eMissingDxfField;
        } else
            fieldsFlags >>= 1;
    }
    mPlaneNormal = planeNormal;
    mNumSides = numSides;
    mTextStyle = textStyle;
    setName(pName);
    acutDelString(pName);
    if (version==1)
    {
        //convert data from old format
        acdbWcs2Ecs(asDblArray(cen3d),asDblArray(cen3d),
            asDblArray(planeNormal),Adesk::kFalse);
        mCenter.set(cen3d.x,cen3d.y);
        mElevation = cen3d.z;

        acdbWcs2Ecs(asDblArray(sp3d),asDblArray(sp3d),
            asDblArray(planeNormal),Adesk::kFalse);
        mStartPoint.set(sp3d.x,sp3d.y);
        assert(mElevation == sp3d.z);
    } else {
        mCenter = cen2d;
        mStartPoint = sp2d;
        mElevation = elevation;
    }
    return es;
}
```

The complete code for the `AsdkPoly` application-defined class can be found in the *samples* directory.

Sample Code for dxfInFields() with Order Dependence

This code sample shows how you could write a `dxfInFields()` function that is order-dependent.

```
Acad::ErrorStatus
AsdkPoly::dxfInFields(AcDbDxfFiler* filer)
{
    assertWriteEnabled();

    if ((AcDbCurve::dxfInFields(filer) != Acad::eOk) ||
        !filer->atSubclassData("AsdkPoly") )
    {
        return filer->filerStatus();
    }
```

```
try
{
    struct resbuf rb;

    // Object Version
    Adesk::Int16 version;
    filer->readItem(&rb);
    if (rb.restype != AcDb::kDxfInt16)
        throw AcDb::kDxfInt16;

    version = rb.resval.rint;
    if (version > VERSION)
        return Acad::eMakeMeProxy;

    if (version == 1)
    {
        AcGePoint3d cent,sp;

        filer->readItem(&rb);
        if (rb.restype != AcDb::kDxfXCoord)
            throw AcDb::kDxfXCoord
            cent = asPnt3d(rb.resval.rpoint);

        filer->readItem(&rb);
        if (rb.restype != AcDb::kDxfXCoord + 1)
            throw AcDb::kDxfXCoord + 1;
        sp = asPnt3d(rb.resval.rpoint);

        filer->readItem(&rb);
        if (rb.restype != AcDb::kDxfInt32)
            throw AcDb::kDxfInt32;
        mNumSides = rb.resval.rlong;

        filer->readItem(&rb);
        if (rb.restype != AcDb::kDxfNormalX)
            throw AcDb::kDxfNormalX
            mPlaneNormal = asVec3d(rb.resval.rpoint);

        filer->readItem(&rb);
        if (rb.restype != AcDb::kDxfText)
            throw AcDb::kDxfText;
        setName(rb.resval.rstring);

        filer->readItem(&rb);
        if (rb.restype != kDxfHardPointerId)
            throw AcDb::kDxfHardPointerId;
        acdbGetObjectId(mTextStyle, rb.resval.rlname);

        // Convert data from old format.
        acdbWcs2Ecs(asDblArray(cent),asDblArray(cent),
            asDblArray(mPlaneNormal),Adesk::kFalse);
        mCenter.set(cent.x,cent.y);
        mElevation = cent.z;
```

```
            acdbWcs2Ecs(asDblArray(sp),asDblArray(sp),
                asDblArray(mPlaneNormal),Adesk::kFalse);
            mStartPoint.set(sp.x,sp.y);
            assert(mElevation == sp.z);
        }
        else if (version == 2)
        {
            filer->readItem(&rb);
            if (rb.restype != AcDb::kDxfXCoord)
                throw AcDb::kDxfXCoord;
            mCenter = asPnt2d(rb.resval.rpoint);

            filer->readItem(&rb);
            if (rb.restype != AcDb::kDxfXCoord + 1)
                throw AcDb::kDxfXCoord + 1;
            mStartPoint = asPnt2d(rb.resval.rpoint);

            filer->readItem(&rb);
            if (rb.restype != AcDb::kDxfInt32)
                throw AcDb::kDxfInt32
                mNumSides = rb.resval.rlong;

            filer->readItem(&rb);
            if (rb.restype != AcDb::kDxfNormalX)
                throw AcDb::kDxfNormalX;
            mPlaneNormal = asVec3d(rb.resval.rpoint);

            filer->readItem(&rb);
            if (rb.restype != AcDb::kDxfText)
                throw AcDb::kDxfText
                setName(rb.resval.rstring);

            filer->readItem(&rb);
            if (rb.restype != AcDb::kDxfHardPointerId)
                throw AcDb::kDxfHardPointerId;
            acdbGetObjectId(mTextStyle, rb.resval.rlname);

            filer->readItem(&rb);
            if (rb.restype != AcDb::kDxfReal)
                throw AcDb::kDxfReal;
            mElevation = rb.resval.rreal;
        }
        else assert(false);
    }
    catch (AcDb::DxfCode code)
    {
        filer->pushBackItem();
        filer->setError(Acad::eInvalidDxfCode,
            "\nError: expected group code %d", code);
        return filer->filerStatus();
    }
}
```

Object References

An object reference can be either hard or soft, and it can be either an ownership reference or a pointer reference. The hard or soft distinction indicates whether the referenced object is essential to the existence of the object that refers to it. A hard reference indicates that an object depends on the referenced object for its survival. A soft reference indicates that an object has some kind of relationship to the referenced object, but it is not an essential one.

An ownership reference dictates how objects are filed. If one object owns another, then whenever the first object is filed out, it takes the owned object with it. Because an object can have only one owner, ownership references are used for nonredundant writing out of the database. In contrast, pointer references are used to express any arbitrary reference between AcDb objects. Pointer references are used for complete (redundant) writing out of the database.

For example, in the following figure, the double lines indicate ownership references. If you follow the double lines, you touch every object in this small database only once. If you also follow the single lines, which represent pointer references, you touch some objects more than once, because multiple objects can point to the same object. To obtain the full "definition" of the `AcDbLine` object, you would need to follow all the hard references, both ownership and pointer (that is, both the single and double solid lines).

Ownership References

If you are creating your own ownership hierarchy, you need to set up the connection between the owner and the owned object. An object cannot have multiple owners.

To create an ownership connection

1 Specify that the owner owns the object.
2 Specify that the object belongs to the owner.

The `AcDbObject` protocol always specifies the link from the owner to the owned object and the backward link from the object to its owner.

The following code illustrates setting up the two-way ownership link between an owner and its contents:

```
// Uses the OwnerDemo class defined in the next example
// (see "ObjectARX Example," below).
//
// Sets pOwner to be the owner of pOwned.
//
void
makeOwner(OwnerDemo* pOwner, AcDbObject* pOwned)
{
    // First let pOwner know it is the owner.  This
    // establishes ownership for filing persistence.
    //
    pOwner->setIdData(pOwned->ojectId());

    // Now set up the backpointer so that the owned
    // object knows who its owner is.
    //
    pOwned->setOwnerId(pOwner->objectId());
}
```

Most commonly used container class members establish the two-way link automatically. For example, the following function call sets the block table record as the owner of the entity, and also adds the entity to the block table record's list of owned entities.

```
blockTableRecord->appendAcDbEntity( ...);
```

Similarly, the `AcDbDictionary::setAt()` function and the `AcDbSymbolTable::add()` function set up two-way links between the owner and its objects in one step.

If you are directly manipulating objects using `entmod()` or `entmake()` in AutoLISP, you first add the owned object to the database using `entmake()`,

then associate its `ads_name` or entity name with the appropriate DXF group code in the owner object representation.

Uses of Ownership

When an object is written to a DXF or DWG file, all objects owned by this object are also written out. The deep clone operation also recursively copies every object owned by the cloned object. See chapter 18, "Deep Cloning." A hard ownership relationship protects the owned object from purge.

Types of Ownership

Owners can be either hard or soft owners of their objects.

Hard Ownership

The following are three examples of hard ownership:

- A database object is a hard owner of its extension dictionary.
- The block table is a hard owner of the model space and paper space block table records (but not the other block table records).
- Extension dictionaries are hard owners of their elements.

Soft Ownership

A soft ownership ID (of type `AcDbSoftOwnershipId`) does not protect the owned object from purge. The following are examples of soft ownership:

- In most cases, symbol tables are soft owners of their elements (exceptions include the block *MODEL_SPACE, *PAPER_SPACE, *PAPER_SPACE0, and layer 0; for these elements, the symbol table maintains a hard reference).
- Dictionaries are soft owners of their entries (but you can flag a dictionary to be a hard owner of its entries).

Building an Ownership Hierarchy

The following example illustrates how to build an ownership hierarchy using ObjectARX functions. The example shows header and source files for a new class, `OwnerDemo`, which illustrates how to create an ownership tree. This class has two data members, a simple integer to represent normal data, and a hard ownership ID data member to hold the object ID of an owned object. Functions are provided for getting and setting the values of both data members. The example also overrides the four required virtual functions: `dwgInFields()`, `dwgOutFields()`, `dxfInFields()`, and `dxfOutFields()`.

The ownership hierarchy is set up in the `createObjs()` routine toward the end of the example. Object A owns object B. Object B owns object C. Object A is added to a dictionary (ASDK_DICT) in the named object dictionary. The `printOut()` and `listTree()` routines print information on the objects in the ASDK_DICT dictionary.

ObjectARX Example

```
// Class declarations
//
class AsdkOwnerDemo : public AcDbObject

// This is a custom object class to demonstrate what is
// necessary to create ownership trees.
//
// To keep it simple, this class has two data members: a
// simple integer to represent normal data, and a hard
// ownership ID data member to hold the object ID of an owned
// object.
//
// Get and set functions are provided for both data members.
//
{
public:
    ACRX_DECLARE_MEMBERS(AsdkOwnerDemo);
    AsdkOwnerDemo(): mIntval(0) {};
    AsdkOwnerDemo(const Adesk::Int16& val): mIntval(val) {};

    Adesk::Int16 intData();
    Acad::ErrorStatus setIntData(const Adesk::Int16&);

    AcDbHardOwnershipId idData();
    Acad::ErrorStatus setIdData(const AcDbHardOwnershipId&);

    Acad::ErrorStatus dwgInFields (AcDbDwgFiler*);
    Acad::ErrorStatus dwgOutFields(AcDbDwgFiler*) const;
    Acad::ErrorStatus dxfInFields (AcDbDxfFiler*);
    Acad::ErrorStatus dxfOutFields(AcDbDxfFiler*) const;
private:
    Adesk::Int16 mIntval;
    AcDbHardOwnershipId mObjId;
};

ACRX_DXF_DEFINE_MEMBERS(AsdkOwnerDemo, AcDbObject,
    AcDb::kDHL_CURRENT, AcDb::kMReleaseCurrent, 0,
    ASDKOWNERDEMO, OWNERSHIP);

// Gets the value of the integer data member.
//
Adesk::Int16
AsdkOwnerDemo::intData()
{
    assertReadEnabled();
    return mIntval;
}
```

```
// Sets the value of the integer data member.
//
Acad::ErrorStatus
AsdkOwnerDemo::setIntData(const Adesk::Int16& val)
{
    assertWriteEnabled();
    mIntval = val;
    return Acad::eOk;
}

// Returns a copy of the ownership ID data member.
//
AcDbHardOwnershipId
AsdkOwnerDemo::idData()
{
    assertReadEnabled();

    return mObjId;
}
// Sets the value of the ownership ID data member.
//
Acad::ErrorStatus
AsdkOwnerDemo::setIdData(const AcDbHardOwnershipId& ownedId)
{
    if (ownedId.asOldId() == 0L) {
        return Acad::eInvalidInput;
    }
    assertWriteEnabled();
    mObjId = ownedId;

    // Now set the backpointer.  A transaction is used for
    // opening the object, so if the object is already
    // open it won't prevent this setting from taking place.
    //
    AcDbObject *pObj;
    AcTransaction *pTrans
        = actrTransactionManager->startTransaction();
    pTrans->getObject(pObj, ownedId, AcDb::kForWrite);
    pObj->setOwnerId(objectId());
    actrTransactionManager->endTransaction();

    return Acad::eOk;
}

// Files data in from a DWG file.
//
Acad::ErrorStatus
AsdkOwnerDemo::dwgInFields(AcDbDwgFiler* filer)
{
    assertWriteEnabled();
    AcDbObject::dwgInFields(filer);

    // For wblock filing we wrote out our owner as a hard
    // pointer Id so now we need to read it in to keep things
    // in sync.
    //
```

```
    if (filer->filerType() == AcDb::kWblockCloneFiler) {
        AcDbHardPointerId id;
        filer->readItem(&id);
    }

    filer->readItem(&mIntval);
    filer->readItem(&mObjId);
    return filer->filerStatus();
}

// Files data out to a DWG file.
//
Acad::ErrorStatus
AsdkOwnerDemo::dwgOutFields(AcDbDwgFiler* filer) const
{
    assertReadEnabled();
    AcDbObject::dwgOutFields(filer);

    // Since objects of this class will be in the Named
    // Objects Dictionary tree and may be hard referenced
    // by some other object, to support wblock we need to
    // file out our owner as a hard pointer Id so that it
    // will be added to the list of objects to be wblocked
    //
    if (filer->filerType() == AcDb::kWblockCloneFiler)
        filer->writeHardPointerId((AcDbHardPointerId)ownerId());

    filer->writeItem(mIntval);
    filer->writeItem(mObjId);
    return filer->filerStatus();
}

// Files data in from a DXF file.
//
Acad::ErrorStatus
AsdkOwnerDemo::dxfInFields(AcDbDxfFiler* filer)
{
    assertWriteEnabled();
    Acad::ErrorStatus es;
    if ((es = AcDbObject::dxfInFields(filer))
        != Acad::eOk)
    {
        return es;
    }

    // Check if we're at the right subclass data marker.
    //
    if (!filer->atSubclassData("AsdkOwnerDemo")) {
        return Acad::eBadDxfSequence;
    }
```

```
    struct resbuf inbuf;
    while (es == Acad::eOk) {
        if ((es = filer->readItem(&inbuf)) == Acad::eOk) {
            if (inbuf.restype == AcDb::kDxfInt16) {
                mIntval = inbuf.resval.rint;
            } else if (inbuf.restype
                == AcDb::kDxfHardOwnershipId)
            {
                acdbGetObjectId(mObjId,
                    inbuf.resval.rlname);
            }
        }
    }

    return filer->filerStatus();
}

// Files data out to a DXF file.
//
Acad::ErrorStatus
AsdkOwnerDemo::dxfOutFields(AcDbDxfFiler* filer) const
{
    assertReadEnabled();
    AcDbObject::dxfOutFields(filer);

    filer->writeItem(AcDb::kDxfSubclass, "AsdkOwnerDemo");
    filer->writeItem(AcDb::kDxfInt16, mIntval);

    // Null object IDs are invalid: don't write them out.
    //
    if (mObjId.asOldId() != 0L) {
        filer->writeItem(AcDb::kDxfHardOwnershipId, mObjId);
    }

    return filer->filerStatus();
}

// Creates an AsdkOwnerDemo object (pObjC) and adds data to
// it.  Then, AsdkOwnerDemo pObjC is created and set to be
// the owner of pObjC.  Next, AsdkOwnerDemo pObjA is created
// and set to own pObjB.  Finally, pObjA is added to a
// dictionary in the named object dictionary.  Technically,
// we could just add pObjA to the named object dictionary
// itself, but that's not appropriate because it would clutter
// up the named object dictionary.
//
void
createObjs()
{
    AcDbObjectId objIdA, objIdB, objIdC;
    AcDbDictionary *pNamedobj;
    AcDbDictionary *pDict = NULL;
    AcDbDatabase *pCurDwg =
        acdbHostApplicationServices()->workingDatabase();
```

```
// Create object C with a dummy integer data value of 3.
//
AsdkOwnerDemo *pObjC = new AsdkOwnerDemo(3);

// Append object C to database without setting an owner.
//
pCurDwg->addAcDbObject(objIdC, pObjC);
pObjC->close();

// Create object B with a dummy integer data value of 2.
//
AsdkOwnerDemo *pObjB = new AsdkOwnerDemo(2);

// Append object B to the database without setting an owner.
//
pCurDwg->addAcDbObject(objIdB, pObjB);

// Now set up ownership for object C.  The
// AsdkOwnerDemo::setIdData() function takes the
// objectId parameter and copies it into the
// AcDbHardOwnershipId data member.  This places the
// object ID in a position to be filed out/in via the
// dwgInFields/dwgOutFields/dxfInFields/dxfOutFields
// member functions.  This constitutes primary
// "ownership."  The AsdkOwnerDemo::setIdData() function
// also calls each owned object's setOwnerId() member
// function to set the backpointer and establish the
// full two-way ownership link.
//
pObjB->setIdData(objIdC);
pObjB->close();

// Create object A with a dummy integer data value of 1.
//
AsdkOwnerDemo *pObjA = new AsdkOwnerDemo(1);

// Next, add objA to a dictionary in the named object
// dictionary.  This will establish ownership for objA,
// set the ownership backlink, and add it to the
// database.
//
pCurDwg->getNamedObjectsDictionary(pNamedobj,
    AcDb::kForWrite);

// Get a pointer to the ASDK_DICT dictionary.  If it
// doesn't exist, then create it and add it to the
// named object dictionary.
//
if (pNamedobj->getAt("ASDK_DICT", (AcDbObject*&) pDict,
    AcDb::kForWrite) == Acad::eKeyNotFound)
{
    pDict = new AcDbDictionary;
    AcDbObjectId DictId;
    pNamedobj->setAt("ASDK_DICT", pDict, DictId);
}
```

```
    pNamedobj->close();

    // Add object A to the ASDK_DICT dictionary.
    //
    pDict->setAt("OBJA", pObjA, objIdA);
    pDict->close();

    // Now set up ownership for object B.
    //
    pObjA->setIdData(objIdB);
    pObjA->close();
}

// The list tree function runs through all objects in the
// ASDK_DICT dictionary, follows their ownership trees, and
// lists out information on all objects in the tree.
//
void
listTree()
{
    AcDbDictionary *pNamedobj;
    AcDbDictionary *pDict;
    acdbHostApplicationServices()->workingDatabase()
        ->getNamedObjectsDictionary(pNamedobj, AcDb::kForWrite);

    // Get a pointer to the ASDK_DICT dictionary.
    //
    pNamedobj->getAt("ASDK_DICT",(AcDbObject*&)pDict,
        AcDb::kForRead);
    pNamedobj->close();

    // Run through the entries and list their backpointers.
    //
    AcDbDictionaryIterator *pDictItr = pDict->newIterator();
    for (; !pDictItr->done(); pDictItr->next()) {
        printOut(pDictItr->objectId());
    }
    pDict->close();
}

// Recursively walks down an ownership tree of AsdkOwnerDemo
// class objects, printing out information on each one.
//
void
printOut(AcDbObjectId id)
{
    AsdkOwnerDemo *pDemo;
    acdbOpenObject((AcDbObject*&)pDemo, id, AcDb::kForRead);
    acutPrintf("\nIntdata: %d  ObjId: %ld  Backpointer:"
        " %ld OwnedObj: %ld", pDemo->intData(),
        (pDemo->objectId()).asOldId(),
        (pDemo->ownerId()).asOldId(),
        (pDemo->idData()).asOldId());
```

```
    // Recursive tree walk
    //
    if ((pDemo->idData()).asOldId() != 0L) {
        printOut(pDemo->idData());
    }
    pDemo->close();
}

// The initialization function is called from acrxEntryPoint()
// during kInitAppMsg case.  This function is used to add commands
// to the command stack and to add classes to the ACRX class
// hierarchy.
//
void
initApp()
{
    acedRegCmds->addCommand("ASDK_OWNERSHIP_COMMANDS",
        "ASDK_CREATE", "CREATE",ACRX_CMD_MODAL, createObjs);
    acedRegCmds->addCommand("ASDK_OWNERSHIP_COMMANDS",
        "ASDK_LISTREE", "LISTREE",ACRX_CMD_MODAL, listTree);

    AsdkOwnerDemo::rxInit();
    acrxBuildClassHierarchy();
}

// The clean up function is called from acrxEntryPoint() during the
// kUnloadAppMsg case.  This function removes this application's
// command set from the command stack and the
// AsdkOwnerDemo class from the ARX runtime class tree.
//
void
unloadApp()
{
    acedRegCmds->removeGroup("ASDK_OWNERSHIP_COMMANDS");

    // Remove the AsdkOwnerDemo class from the ACRX runtime
    // class hierarchy.  If this is done while the database is
    // still active, it should cause all objects of class
    // AsdkOwnerDemo to be turned into proxies.
    //
    deleteAcRxClass(AsdkOwnerDemo::desc());
}
```

```
// ObjectARX entry point
//
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
    }
    return AcRx::kRetOK;
}
```

Pointer References

Your custom class may also contain hard or soft pointer references to other objects in the database. A pointer is a one-way link (that is, there is no information in the referenced object that indicates the source of the pointer). An object can point to, or be pointed to by, any number of other objects.

Hard Pointers

A hard pointer reference protects an object from purge. For example, an entity contains a hard pointer reference to a layer. Therefore, you can't purge a layer that is pointed to by one or more entities. When a new database is written out from an existing one (for example, in a WBLOCK operation), all hard pointers are copied into the new database.

Other examples of hard pointer references

- A leader entity contains a hard pointer reference to a dimension style.
- A text entity contains a hard pointer reference to a text style.
- A dimension entity contains a hard pointer reference to a dimension style.
- An mline entity has a hard pointer reference to an mline style.

Soft Pointers

A soft pointer is simply a pointer to an object. It does not protect the referenced object from purge.

Examples of soft pointer references

- Xdata references are soft pointers.
- Persistent reactors are soft pointers.

If you use a soft pointer to refer to an object, you should check that the object still exists before you open it.

Long Transaction Issues for Custom Objects

Long Transactions are defensive against common problems resulting from objects that should not be cloned, or inter-object references that are not handled. If the LongTransactionManager (LTM) finds that it must clone a filtered class object to complete a long transaction `checkOut()` or `checkIn()`, it will abort the entire operation. If it finds an `AcDbSoftPointerId` or `AcDbHardPointerId` that is not in the cloning IdMap, it will also abort.

Applications that need to prevent their objects from being included as cloned in long transactions need to register those objects using the `AcApLongTransactionManager::addClassFilter()` function. `AcDbProxyEntity` and `AcDbProxyObject` are always filtered, so when the application is not present, all of its objects will be filtered automatically.

Wblock cloning handles all hard pointer references, but deep cloning does not require either type of reference to be mapped. Both of these cloning types are used in long transactions, depending on the type of transaction it is. If an application uses either of these types of references, or xdata handles, then its objects will be rejected from long transactions, unless the application takes extra steps to handle the references. This means that if the application is not loaded, then its objects and references will automatically be prevented from participating in long transactions and any data should be preserved in its absence.

Use the long transaction and deep clone notifications to intercept the cloning of their object and references, and to add whatever object cloning or mapping is needed. See the deep clone notification documentation and samples for more information on this.

If an object with a soft pointer reference is cloned (or a hard pointer reference in deep clone), the application must make sure that the reference ID is in the

IdMap, either as a mapped ID pair, or a cloned ID pair. Mappings are usually used when objects refer to some common dictionary that the application maintains within the drawing. In deep clone, the mapping may consist of an IdPair where the key == value. In wblock clone between drawings, the IdPair would map one database's dictionary with the other database's dictionary. The reference is cloned by the application using either `deepClone()` or `wblockClone()` from the notification callback.

Taking these steps will guarantee "transitive closure." To ensure that a set of objects that refer to each other can be checked out, and then checked back in again, without breaking the object's relationships, all associated objects are checked out together. For example, if any boundary or the associative hatch itself is passed into `checkOut()`, the hatch code adds all of the boundary objects to the list of objects to be checked out. This is how it achieves transitive closure. If this did not happen, the LTM would find the hatch's soft pointer IDs to its boundaries. If it found that a boundary object so referenced was missing from the cloning, the long transaction would be aborted. If it did not do this, even if no changes were made to the checked out hatch, the original hatch would lose its associativity on check in.

Sometimes, there are known references that do not need to be resolved. One situation would be an object that keeps track of all the entities that use it. For example, block table records keep a list of all the block references that use them. It is correct to only check out one of the references, so you must let the long transaction mechanism know that the rest of the references do not need to be cloned. There are several ways this can be done.

Here are some examples:

- If the application knows about which objects are referenced but will not be cloned—at `beginWblockObjects()`, `beginDeepClone()`, or `beginCheckOut()` notification—they can add the object ID of the referenced object to the `IdMap` for the cloning. The recommended approach is to set the value to `NULL`, and the `idPair` as not cloned. For example
  `idMap.assign(idPair(id, AcDbObjectId::kNull, kFalse);`

  If the object needs to be cloned later, the `idPair` will be changed accordingly.
- The above mapping can also be done from within the object's `wblockClone()` method, if that has already been overridden.
- If the reference is a data member of the object, which is filed out using `dwgOutFields()`, then it may be possible to avoid the long transaction validity test. The test is done by filing out the IDs using a `kIdFiler` type of filer. To avoid the test, do not file out the IDs that do not need to be cloned, during this type of filing. However, do not hold any ownership

IDs out of this filing, or other features that use this filer, like partial save and load, may not properly handle your objects. The only safe IDs to withhold from this filer are `AcDbSoftPointerId` and `AcDbHardPointerId` objects.

- If the `ID` is actually in a persistent reactor, it is possible to find it using the reactor iterator. Here's an example of how a dictionary object finds and adds its ID to the `IdMap` during `beginWblockClone()` notification.

```
beginWblockClone(..., AcDbIdMapping& idMap)
{
    ...
    AcDbDictionaryIterator* pIter = pDict->newIterator();
    AcDbObject* pObj;
    for ( ; !pIter->done(); pIter->next()) {
        acdbOpenObject(pObj, pIter->objectId(), kForRead);
        AcDbVoidPtrArray* pReactors = pObj->reactors();
        void* pReactor;
        AcDbObjectId rId;
        MyReactor* pMyReactor;
        if (pReactors) {
            for (int i = 0; i < pReactors->length(); i++) {
                pReactor = pReactors->at(i);
                if (acdbIsPersistentReactor(pReactor)) {
                    rId = acdbPersistentReactorObjectId(pReactor);
                    if (acdbOpenObject(pMyReactor, rId, kForRead)
                            == eOk) {
                        pMyReactor->close();
                        AcDbIdPair idPair(rId,
                            AcDbObjectId::kNull, kFalse);
                        idMap.assign(idPair);
                    }
                }
            }
        }
        pObj->close();
    }
    delete pIter;
    pDict->close();
}
```

Purge

The purge mechanism allows you to erase unused objects in the database. If an object has a hard owner or pointer reference, it cannot be purged. The `purge()` function of `AcDbDatabase` is invoked on the set of objects specified in the ID array:

```
AcDbDatabase::purge(AcDbObjectIdArray &idArray);
```

The `purge()` function returns in the same ID array the IDs of the objects that can be purged (that is, that have no hard references to them). Once you have this array of object IDs, you are responsible for erasing the objects.

When a drawing is loaded, AutoCAD goes through the database and purges unreferenced anonymous blocks and nested xref blocks. These blocks are erased when the drawing file is closed. If you create any anonymous blocks between the open and close of a drawing, they will be purged without your knowledge unless you protect them by calling the standalone function `acdbSetReferenced()`. This purging occurs even if the objects have hard references to them.

Undo and Redo

There are two basic ways of recording the state for an undo operation. The automatic undo mechanism, the default, lets the system copy the object's complete state by calling the object's `dwgOutFields()` function with the undo filer. An alternative mechanism, referred to as the partial undo mechanism, requires more programming effort but enables you to write out and read in only the specific information regarding the particular modifications that were made to the object.

Every modification function for your new class (for example, any `set()` function) is required to call the `assertWriteEnabled()` function, which checks that the object is write-enabled. If the value of the `autoUndo` parameter for this function is `kTrue`, the object is recorded for undo. When the object modification is complete and the object is closed, the contents of the filer are saved into an undo file. For a given class, some modification functions can use the auto undo mechanism and others can implement a partial undo mechanism. The partial undo mechanism is useful if the modification involves a small amount of data.

When an UNDO command is invoked and an auto undo operation was performed, AutoCAD invokes `dwgInFields()` on the object, thus reading in the contents of the undo file.

Automatic Undo

The `assertWriteEnabled()` function has the following signature:

```
void assertWriteEnabled(
    Adesk::Boolean autoUndo = Adesk::kTrue,
    Adesk::Boolean recordModified = Adesk::kTrue);
```

When a modification function calls `assertWriteEnabled()`, it first checks the value of the `recordModified` parameter. If `recordModified` is `kFalse`, no undo recording is performed. If `recordModified` is `kTrue`, it next checks the `autoUndo` parameter, which specifies whether an auto undo operation should be performed.

If `autoUndo` is `kTrue` (the default), the full object state is automatically written to the object's undo filer. If you specify `kFalse` for `autoUndo`, no information is recorded. AutoCAD assumes that your modification function will take care of recording the changed object state to the object's undo filer.

Even if you plan to implement a partial undo mechanism for your class, you can rely on automatic undo in the first stages of development.

Partial Undo

It is up to the implementor of a new class to decide whether to implement a partial undo mechanism for certain modification functions of the class. If only a small portion of an object's state is typically modified in a particular member function, using partial undo can yield substantial performance benefits. However, if your object state is small (512 bytes or less), it is probably not worth the effort to implement your own partial undo recording and restoring scheme.

If your modification function records a partial object state, you must implement the `applyPartialUndo()` function for your class so that the data can also be restored selectively. See "Restoring State" on page 326.

Recording State

To record only part of an object's state, specify `kFalse` for the `autoUndo` parameter, and then use the `undoFiler::writeItem()` function (or another `write`*xxx*`()` function) to save the relevant information in the undo file.

The `setNumSides()` function of `AsdkPoly` is a typical example of a modification function. Because `assertWriteEnabled()` specifies `kFalse` for `autoUndo`,

the class assumes the responsibility of recording relevant parts of the object's state. First, the modification function must record the class descriptor object so that derived classes can check and let this class process its partial undo data if necessary.

```
undoFiler()->writeItem((long)AsdkPoly::desc());
```

Then the modification function needs to indicate the type of action, followed by the data. In this example, the type of operation is `kSetNumSides` and the data is `mNumSides`.

```
Acad::ErrorStatus
AsdkPoly::setNumSides(int numSides)
{
    assertWriteEnabled(Adesk::kFalse, Adesk::kTrue);
    if (numSides<3)
        return Acad::eInvalidInput;

    if (mNumSides == numSides)
        return Acad::eOk;

    // There are situations under which AutoCAD doesn't
    // want to do undo recording, so it won't create an
    // undo filer. Check for the existence of the filer
    // before starting to write into it.
    //
    AcDbDwgFiler *pFiler = NULL;
    if ((pFiler = undoFiler()) != NULL) {
        undoFiler()->writeItem((long)AsdkPoly::desc());
        undoFiler()->writeItem((Adesk::Int16)kSetNumSides);
        undoFiler()->writeItem((Adesk::Int32)mNumSides);
    }
    mNumSides = numSides;
    return Acad::eOk;
}
```

Once an object has performed an auto undo operation, which records its full state, additional requests for auto undo are ignored.

Restoring State

If you specified `kFalse` for `autoUndo`, the object's `applyPartialUndo()` function is called when the UNDO command is invoked. The `applyPartialUndo()` function is a virtual function on `AcDbObject`. Derived classes can implement this function to interpret the class-specific information stored by the undo filer and read it in. The `applyPartialUndo()` function must ensure that your class performed the modification. If not, it must super-message, as shown in the following example.

If you are implementing a partial undo mechanism, be sure to call the following function so that no recording happens by default.

```
assertWriteEnabled(kFalse, kFalse);
```

As an example, here is `AsdkPoly`'s `applyPartialUndo()` function:

```
Acad::ErrorStatus
AsdkPoly::applyPartialUndo(AcDbDwgFiler* filer,
                           AcRxClass*    classObj)
{
    // The first thing to check is whether the class matches
    // ours. If it doesn't, we call the base class's
    // applyPartialUndo(); hopefully, one of them will
    // take care of it.
    //
    if (classObj != AsdkPoly::desc())
        return AcDbCurve::applyPartialUndo(filer, classObj);

    // Read the op-code and call the appropriate "set"
    // method to undo what was done. The "set" does the
    // filing again for redo.
    //
    Adesk::Int16 shortCode;
    filer->readItem(&shortCode);

    PolyOpCodeForPartialUndo code;
    code = (PolyOpCodeForPartialUndo)shortCode;

    Adesk::UInt32 value32;
    switch (code) {
    case kSetNumSides:
        filer->readItem(&value32);
        AOK(setNumSides(value32));
        break;
    default:
        assert(Adesk::kFalse);
        break;
    }
    return Acad::eOk;
}
```

Redo

When the undo operation undoes your work, it also records the current state in preparation for a redo operation. This recording for redo requires no further work on your part, because it uses the same filing mechanism as the undo operation, calling the object's `dwgOutFields()` function to record the object's state.

If you implement a partial undo for your modification function, you are responsible for recording for redo in your undo operation. This is usually accomplished by calling the appropriate `set()` functions. When your `set()` function is called, `assertWriteEnabled()` is invoked, which records the data for undo.

subErase, subOpen, subClose, and subCancel

The `erase()`, `open()`, `close()`, and `cancel()` functions all have corresponding virtual functions beginning with the prefix *sub*. You can override these subsidiary functions to provide extra functionality for derived classes. The subsidiary function is invoked by the nonvirtual "master" function. For example, `erase()` calls `subErase()`. The signature for `subErase()` is as follows:

```
virtual Acad::ErrorStatus
subErase(Adesk::Boolean pErasing);
```

To override a subsidiary function

1. Validate your surroundings. For example, if your object has a hard pointer reference to another object and your object is being unerased, you can check that the object you refer to still exists. If there are problems, immediately return an appropriate error status and don't pass the message up, because your bad error status will effectively kill the operation.
2. If everything is OK, then invoke the `Your Parent::subErase()` function. Examine its result. If it does not return `eOK`, then return.
3. If everything is OK, then perform your actions.

It is best not to change any state in a subsidiary function. If you must change state, then try to change it after invoking the parent class implementation of the same function (in case an error code is returned). If you must change state before invoking the parent class function, then be prepared to reverse it if the parent class returns a bad status.

The following example shows the implementation of the `subErase()` function that is called when an object is erased. The `subErase()` function checks for hard pointer references to other objects and erases them as well.

```
class AsdkEllipse : public AcDbEllipse
// This class extends AcDbEllipse by adding in functionality
// to store a dynamic array of hard pointer object IDs.
//
// The subErase() member function has been overridden and
// implemented, so when an object of this class is
// erased, the objects pointed to by the hard pointer IDs
// stored within the object will also be erased.
//
```

```
{
public:

    ACRX_DECLARE_MEMBERS(AsdkEllipse);
    AsdkEllipse() {};
    AsdkEllipse(const AsdkEllipse&);
    AsdkEllipse(const AcDbObjectIdArray& ellipses)
        : mEllipseIds(ellipses) {};
    AsdkEllipse(const AcGePoint3d&  center,
        const AcGeVector3d& unitNormal,
        const AcGeVector3d& majorAxis,
        double radiusRatio,
        double startAngle = 0.0,
        double endAngle = 6.28318530717958647692);
    AsdkEllipse(const AcDbObjectIdArray&  ellipses,
        const AcGePoint3d&  center,
        const AcGeVector3d& unitNormal,
        const AcGeVector3d& majorAxis,
        double radiusRatio,
        double startAngle = 0.0,
        double endAngle   = 6.28318530717958647692);
    AcDbObjectId ellipseId(unsigned short which);
    Acad::ErrorStatus setEllipseId(
        const AcDbObjectId& objId, unsigned short which);
    Acad::ErrorStatus setEllipseIds(
        const AcDbObjectIdArray& Ids);
    Acad::ErrorStatus appendId(const AcDbObjectId& objId);
    Acad::ErrorStatus appendIds(
        const AcDbObjectIdArray& objIds);
    inline Adesk::Boolean removeId(
        const AcDbObjectId& objId);

    // AcDbObject overrides.
    //
    virtual Acad::ErrorStatus subErase(
        Adesk::Boolean pErasing);
    virtual Acad::ErrorStatus dwgInFields(
        AcDbDwgFiler* filer);
    virtual Acad::ErrorStatus dwgOutFields(
        AcDbDwgFiler* filer) const;
    virtual Acad::ErrorStatus dxfInFields(
        AcDbDxfFiler* filer);
    virtual Acad::ErrorStatus dxfOutFields(
        AcDbDxfFiler* filer) const;
    virtual Acad::ErrorStatus wblockClone(
        AcRxObject* pOwnerObject,
        AcDbObject*& pClonedObject,
        AcDbIdMapping& idMap,
        Adesk::Boolean isPrimary = Adesk::kTrue) const;

    // AcDbEntity overrides.
    //
    virtual void list() const;
    // AcRxObject overrides.
    //
    virtual AcRxObject* clone() const;
```

```
private:
    AcDbObjectIdArray mEllipseIds;
    static int mInFlux;  // == 1 when first object's
                         //    subErase is active.
};
ACRX_DXF_DEFINE_MEMBERS(AsdkEllipse, AcDbEllipse,
    AcDb::kDHL_CURRENT, AcDb::kMReleaseCurrent, 0, ASDKELLIPSE,
    REFERENC);

// Static class data member definition.
//
int AsdkEllipse::mInFlux = Adesk::kFalse;
AsdkEllipse::AsdkEllipse(const AsdkEllipse& master)
{
    set(master.center(), master.normal(),
        master.majorAxis(), master.radiusRatio(),
        master.startAngle(), master.endAngle());

    mEllipseIds = master.mEllipseIds;
}

AsdkEllipse::AsdkEllipse(const AcGePoint3d&  center,
    const AcGeVector3d& unitNormal,
    const AcGeVector3d& majorAxis,
    double              radiusRatio,
    double              startAngle,
    double              endAngle) :
    AcDbEllipse(center, unitNormal, majorAxis, radiusRatio,
    startAngle, endAngle)
{ }

AsdkEllipse::AsdkEllipse(const AcDbObjectIdArray&  ellipses,
    const AcGePoint3d&  center,
    const AcGeVector3d& unitNormal,
    const AcGeVector3d& majorAxis,
    double              radiusRatio,
    double              startAngle,
    double              endAngle) :
    AcDbEllipse(center, unitNormal, majorAxis, radiusRatio,
    startAngle, endAngle), mEllipseIds(ellipses)
{ }

AcDbObjectId
AsdkEllipse::ellipseId(unsigned short which)
{
    assertReadEnabled();
    if  (which > mEllipseIds.length())
        return AcDbObjectId::kNull;
    return mEllipseIds[which];
}

Acad::ErrorStatus
AsdkEllipse::setEllipseId(const AcDbObjectId& objId,
    unsigned short which)
{
```

```
    assertWriteEnabled();
    if  (which > mEllipseIds.length())
        return Acad::eInvalidIndex;
    mEllipseIds[which] = objId;
    return Acad::eOk;
}

Acad::ErrorStatus
AsdkEllipse::setEllipseIds(const AcDbObjectIdArray& objIds)
{
    assertWriteEnabled();
    if (objIds.length() == 0)
        return Acad::eNullObjectId;
    mEllipseIds = objIds;
    return Acad::eOk;
}

Acad::ErrorStatus
AsdkEllipse::appendId(const AcDbObjectId& objId)
{
    assertWriteEnabled();
    if  (objId == AcDbObjectId::kNull)
        return Acad::eNullObjectId;
    mEllipseIds.append(objId);
    return Acad::eOk;
}

Acad::ErrorStatus
AsdkEllipse::appendIds(const AcDbObjectIdArray& objIds)
{
    assertWriteEnabled();
    if  (objIds.length() == 0)
        return Acad::eNullObjectId;
    mEllipseIds.append(objIds);
    return Acad::eOk;
}

inline Adesk::Boolean
AsdkEllipse::removeId(const AcDbObjectId& objId)
{
    assertWriteEnabled();
    return mEllipseIds.remove(objId);
}

// This implementation of subErase opens and erases all
// objects that this entity has hard pointer references
// to.  The effect is that when one AsdkEllipse is erased,
// all the others it has hard pointers to also erase as
// a "group".
//
```

```
Acad::ErrorStatus
AsdkEllipse::subErase(Adesk::Boolean pErasing)
{
    Acad::ErrorStatus es = AcDbEllipse::subErase(pErasing);
    if (es != Acad::eOk)
        return es;

    if (mInFlux == Adesk::kFalse) {
        mInFlux = Adesk::kTrue;
        AsdkEllipse *pEllipse;
        int es;
        for (int i = 0; i < mEllipseIds.length(); i++) {
            es = acdbOpenObject(pEllipse, mEllipseIds[i],
                AcDb::kForWrite, Adesk::kTrue);
            if (es != Acad::eOk)
                continue;
                    pEllipse->erase(pErasing);
            pEllipse->close();
        }
        mInFlux = Adesk::kFalse;
    }
    return Acad::eOk;
}

Acad::ErrorStatus
AsdkEllipse::dwgInFields(AcDbDwgFiler* filer)
{
    assertWriteEnabled();
    AcDbEllipse::dwgInFields(filer);

    mEllipseIds.setLogicalLength(0);

    int idCount;
    filer->readInt32((long*)&idCount);

    AcDbHardPointerId objId;
    for (int i = 0; i < idCount; i++) {
        filer->readItem(&objId);
        mEllipseIds.append(objId);
    }
    return filer->filerStatus();
}

Acad::ErrorStatus
AsdkEllipse::dwgOutFields(AcDbDwgFiler* filer) const
{
    assertReadEnabled();
    AcDbEllipse::dwgOutFields(filer);
    filer->writeInt32(mEllipseIds.length());

    for (int i = 0; i < mEllipseIds.length(); i++) {
        filer->writeHardPointerId(mEllipseIds[i]);
    }
    return filer->filerStatus();
}
```

```
Acad::ErrorStatus
AsdkEllipse::dxfInFields(AcDbDxfFiler* filer)
{
    assertWriteEnabled();
    Acad::ErrorStatus es = AcDbEllipse::dxfInFields(filer);
    if (es != Acad::eOk) {
        return es;
    }
    // Check to see if we're at the right subclass data
    // marker.
    //
    if (!filer->atSubclassData("AsdkEllipse")) {
        return Acad::eBadDxfSequence;
    }

    struct resbuf inbuf;
    AcDbObjectId objId;
    int idCount;
    filer->readItem(&inbuf);
    if (inbuf.restype == AcDb::kDxfInt32) {
        idCount = inbuf.resval.rint;
    } else {
        filer->pushBackItem();
        filer->setError(Acad::eInvalidDxfCode,
            "\nError: expected group code %d",
             AcDb::kDxfInt32);
        return filer->filerStatus();
    }
    for (int i = 0; i < idCount; i++) {
        es = filer->readItem(&inbuf);
        if (es != Acad::eOk) {
            filer->setError(Acad::eMissingDxfField,
                "\nError: expected more group code %d's",
                AcDb::kDxfHardPointerId);
            return filer->filerStatus();
        }
        if (inbuf.restype == AcDb::kDxfHardPointerId) {
            acdbGetObjectId(objId, inbuf.resval.rlname);
            mEllipseIds.append(objId);
        } else {
            filer->pushBackItem();
            filer->setError(Acad::eInvalidDxfCode,
                "\nError: expected group code %d",
                AcDb::kDxfHardPointerId);
            return filer->filerStatus();
        }
    }
    return filer->filerStatus();
}
```

```
Acad::ErrorStatus
AsdkEllipse::dxfOutFields(AcDbDxfFiler* filer) const
{
    assertReadEnabled();

    AcDbEllipse::dxfOutFields(filer);
    filer->writeItem(AcDb::kDxfSubclass, "AsdkEllipse");
    filer->writeInt32(AcDb::kDxfInt32,
        mEllipseIds.length());

    for (int i = 0; i < mEllipseIds.length(); i++) {
        filer->writeObjectId(AcDb::kDxfHardPointerId,
            mEllipseIds[i]);
    }
    return filer->filerStatus();
}

void
AsdkEllipse::list() const
{
    assertReadEnabled();
    AcDbEllipse::list();
    acutPrintf("\nClass:\t%s", isA()->name());
    for (int i = 0; i < mEllipseIds.length(); i++) {
        acutPrintf("\nReferenceId[%d]:\t%ld", i,
            (mEllipseIds[i]).asOldId());
    }
}

// Called whenever an object of this class is dragged,
// moved, stretched, rotated, etc. so be careful what
// this function is made to do.
//
AcRxObject*
AsdkEllipse::clone() const
{
    assertReadEnabled();
    return new AsdkEllipse(*this);
}
Acad::ErrorStatus
AsdkEllipse::wblockClone(
    AcRxObject*    pOwnerObject,
    AcDbObject*&   pClonedObject,
    AcDbIdMapping& idMap,
    Adesk::Boolean isPrimary) const
{
    assertReadEnabled();
    static AcDbObjectId btr, pspace = AcDbObjectId::kNull;
    AcTransaction *pTrans = NULL;
    pClonedObject = NULL;
    if (pspace == AcDbObjectId::kNull) {
        AcDbBlockTable *pTable;
        database()->getSymbolTable(pTable, AcDb::kForRead);
        pTable->getAt(ACDB_PAPER_SPACE, pspace);
        pTable->close();
    }
```

```
    if (   idMap.deepCloneContext() == AcDb::kDcXrefBind
        && ownerId() == pspace)
        return Acad::eOk;
    // Have we already done this entity ?
    //
    AcDbIdPair idPair(objectId(), (AcDbObjectId)NULL,
        Adesk::kTrue);
    if (idMap.compute(idPair) == TRUE && idPair.value()
        != NULL)
    {
        pClonedObject = NULL;
        return Acad::eOk;
    }
    AcDbBlockTableRecord *pBTR
        = AcDbBlockTableRecord::cast(pOwnerObject);
    if (pBTR != NULL) {
        if (isPrimary == Adesk::kTrue)
            btr = pBTR->objectId();
        else
            btr = AcDbObjectId::kNull;
    } else if (btr != AcDbObjectId::kNull) {
        pTrans = actrTransactionManager->startTransaction();
        pTrans->getObject((AcDbObject*&)pBTR, btr,
            AcDb::kForWrite);
        pOwnerObject = pBTR;
    }
    Acad::ErrorStatus es
        = AcDbEllipse::wblockClone(pOwnerObject,
        pClonedObject, idMap, btr != AcDbObjectId::kNull);
            if (pTrans)
                actrTransactionManager->endTransaction();
    acutPrintf("\nWblockClone error status: %s",
                acadErrorStatusText(es));
    return Acad::eOk;
}
void
createEllipses()
{
    const ellipseCount = 10;
    AsdkEllipse *pEllipse;
    pEllipse = new AsdkEllipse(AcGePoint3d(4.0, 4.0, 0.0),
        AcGeVector3d(0.0, 0.0, 1.0),
        AcGeVector3d(2.0, 0.0, 0.0),
        0.5);
    AcDbVoidPtrArray ellipses;
    ellipses.append(pEllipse);

    // Now use the getTransformedCopy() function with a
    // scaling matrix (in X & Y only) to create new
    // AsdkEllipses, each 0.5 units larger than the last in
    // the X & Y direction,  but identical in the Z
    // direction.  This would be similar to the
    // getOffsetCurves() function,  but that function
    // returns AcDbSpline entities instead of AcDbEllipses.
    //
```

```
    double j = 1.1;
    AcGeMatrix3d scale;
    for (int i = 0; i < ellipseCount; i++, j += 0.1) {
        scale.setToScaling(j, pEllipse->center());
        scale.entry[2][2] = 1.0;  // Z scaling == 1

        // getTransformed copy uses this->clone() to create
        // a new object, which the ent pointer is assigned
        // to point to. Therefore, ent should NOT point to an
        // existing entity or there will be a memory leak!
        //
        // Since this->clone() is used, the AsdkEllipse class
        // must override this member function to
        // be sure that an AsdkEllipse is created instead
        // of just an AcDbEllipse.
        //
        AsdkEllipse *pNextEllipse;
        ((AsdkEllipse*)ellipses[0])->getTransformedCopy(
            scale, (AcDbEntity*&)pNextEllipse);
        ellipses.append(pNextEllipse);
    }
    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);
    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);
    pBlockTable->close();

    AcDbObjectIdArray ellipseIds;
    AcDbObjectId tempId;
    for (i = 0; i < ellipses.length(); i++) {
        pBlockTableRecord->appendAcDbEntity(tempId,
            (AsdkEllipse*)ellipses[i]);
        ellipseIds.append(tempId);
    }
    pBlockTableRecord->close();
    // Set up the hard pointers and close the ellipses.
    //
    for (i = 0; i < ellipses.length(); i++) {
        // Add in all the IDs.
        //
        ((AsdkEllipse*)ellipses[i])
            ->setEllipseIds(ellipseIds);

        // Now remove the object ID of the "*this" ellipse
        // so it doesn't reference itself.
        //
        ((AsdkEllipse*)ellipses[i])->removeId(
            ((AsdkEllipse*)ellipses[i])->objectId());
        ((AsdkEllipse*)ellipses[i])->close();
    }
}
```

```
void
initApp()
{
    acedRegCmds->addCommand("ASDK_ELLIPSES",
        "ASDK_ELLIPSES", "ELLIPSES",
        ACRX_CMD_MODAL, createEllipses);

    AsdkEllipse::rxInit();
    acrxBuildClassHierarchy();
}

void
unloadApp()
{
    acedRegCmds->removeGroup("ASDK_ELLIPSES");

    // Remove the AsdkEllipse class from the ACRX runtime
    // class hierarchy. If this is done while the database is
    // still active, it should cause all objects of class
    // AsdkEllipse to be turned into proxies.
    //
    deleteAcRxClass(AsdkEllipse::desc());
}

extern "C" AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
    }
    return AcRx::kRetOK;
}
```

Example of a Custom Object Class

The following sections show the header and source files for a custom class, `AsdkMyClass`, which is derived from `AcDbObject`. This class stores a single integer value, which can be set and queried with its `setData()` and `getData()` functions. It also implements `dwgInFields()`, `dwgOutFields()`, `dxfInFields()`, and `dxfOutFields()` functions for filing. It is written to and read from file, so its source file uses the `ACRX_DXF_DEFINE_MEMBERS` macro.

Header File

The following code shows the class declaration for the new class `AsdkMyClass` derived from `AcDbObject`.

```
class AsdkMyClass : public AcDbObject
//
// This class demonstrates custom objects.
//
// To keep it simple, this class has a single integer data
// member.  Get and set functions are provided for this
// data member.
//
{
public:
    ACRX_DECLARE_MEMBERS(AsdkMyClass);
    AsdkMyClass(): mIntval(0) {};
    AsdkMyClass(const Adesk::Int16& val): mIntval(val) {};
    Acad::ErrorStatus         getData     (Adesk::Int16&);
    Acad::ErrorStatus         setData     (Adesk::Int16);
    virtual Acad::ErrorStatus dwgInFields (AcDbDwgFiler*);
    virtual Acad::ErrorStatus dwgOutFields(AcDbDwgFiler*)
        const;
    virtual Acad::ErrorStatus dxfInFields (AcDbDxfFiler*);
    virtual Acad::ErrorStatus dxfOutFields(AcDbDxfFiler*)
        const;
private:
    Adesk::Int16 mIntval;
};
```

Source File

The following code shows the implementation for the new class `AsdkMyClass`:

```
ACRX_DXF_DEFINE_MEMBERS(AsdkMyClass, AcDbObject,
    AcDb::kDHL_CURRENT, AcDb::kMReleaseCurrent, 0,
    ASDKMYCLASS, SAMP2);
```

```
// Gets the value of the integer data member.
//
Acad::ErrorStatus
AsdkMyClass::getData(Adesk::Int16& val)
{
    // Tells AutoCAD a read operation is taking place.
    //
    assertReadEnabled();
    val = mIntval;
    return Acad::eOk;
}

// Sets the value of the integer data member.
//
Acad::ErrorStatus
AsdkMyClass::setData(Adesk::Int16 val)
{
    // Triggers openedForModify notification.
    //
    assertWriteEnabled();
    mIntval = val;
    return Acad::eOk;
}

// Files data in from a DWG file.
//
Acad::ErrorStatus
AsdkMyClass::dwgInFields(AcDbDwgFiler* pFiler)
{
    assertWriteEnabled();
    AcDbObject::dwgInFields(pFiler);
    // For wblock filing we wrote out our owner as a hard
    // pointer ID so now we need to read it in to keep things
    // in sync.
    //
    if (pFiler->filerType() == AcDb::kWblockCloneFiler) {
        AcDbHardPointerId id;
        pFiler->readItem(&id);
    }
    pFiler->readItem(&mIntval);
    return pFiler->filerStatus();
}

// Files data out to a DWG file.
//
```

```
Acad::ErrorStatus
AsdkMyClass::dwgOutFields(AcDbDwgFiler* pFiler) const
{
    assertReadEnabled();
    AcDbObject::dwgOutFields(pFiler);
    // Since objects of this class will be in the Named
    // Objects Dictionary tree and may be hard referenced
    // by some other object, to support wblock we need to
    // file out our owner as a hard pointer ID so that it
    // will be added to the list of objects to be wblocked.
    //
    if (pFiler->filerType() == AcDb::kWblockCloneFiler)
        pFiler->writeHardPointerId((AcDbHardPointerId)ownerId());
    pFiler->writeItem(mIntval);
    return pFiler->filerStatus();
}

// Files data in from a DXF file.
//
Acad::ErrorStatus
AsdkMyClass::dxfInFields(AcDbDxfFiler* pFiler)
{
    assertWriteEnabled();
    Acad::ErrorStatus es;
    if ((es = AcDbObject::dxfInFields(pFiler))
        != Acad::eOk)
    {
        return es;
    }
    // Check if we're at the right subclass getData marker.
    //
    if (!pFiler->atSubclassData("AsdkMyClass")) {
        return Acad::eBadDxfSequence;
    }
    struct resbuf inbuf;
    while (es == Acad::eOk) {
        if ((es = pFiler->readItem(&inbuf)) == Acad::eOk) {
            if (inbuf.restype == AcDb::kDxfInt16) {
                mIntval = inbuf.resval.rint;
            }
        }
    }
    return pFiler->filerStatus();
}

// Files data out to a DXF file.
//
Acad::ErrorStatus
AsdkMyClass::dxfOutFields(AcDbDxfFiler* pFiler) const
{
    assertReadEnabled();
    AcDbObject::dxfOutFields(pFiler);
    pFiler->writeItem(AcDb::kDxfSubclass, "AsdkMyClass");
    pFiler->writeItem(AcDb::kDxfInt16, mIntval);
    return pFiler->filerStatus();
}
```

```
// This function creates two objects of class AsdkMyClass.
// It fills them in with the integers 1 and 2, and then adds
// them to the dictionary associated with the key ASDK_DICT.
// If this dictionary doesn't exist, it is created and added
// to the named object dictionary.
//
void
createDictionary()
{
    AcDbDictionary *pNamedobj;
    acdbHostApplicationServices()->workingDatabase()->
        getNamedObjectsDictionary(pNamedobj, AcDb::kForWrite);

    // Check to see if the dictionary we want to create is
    // already present. If not, create it and add
    // it to the named object dictionary.
    //
    AcDbDictionary *pDict;
    if (pNamedobj->getAt("ASDK_DICT", (AcDbObject*&) pDict,
        AcDb::kForWrite) == Acad::eKeyNotFound)
    {
        pDict = new AcDbDictionary;
        AcDbObjectId DictId;
        pNamedobj->setAt("ASDK_DICT", pDict, DictId);
    }
    pNamedobj->close();
    if (pDict) {

        // Create new objects to add to the new dictionary,
        // add them, then close them.
        //
        AsdkMyClass *pObj1 = new AsdkMyClass(1);
        AsdkMyClass *pObj2 = new AsdkMyClass(2);
        AcDbObjectId rId1, rId2;
        pDict->setAt("OBJ1", pObj1, rId1);
        pDict->setAt("OBJ2", pObj2, rId2);
        pObj1->close();
        pObj2->close();
        pDict->close();
    }
}

// Opens the dictionary associated with the key ASDK_DICT
// and iterates through all its entries, printing out the
// integer data value in each entry.
//
```

```
void
iterateDictionary()
{
    AcDbDictionary *pNamedobj;
    acdbHostApplicationServices()->workingDatabase()
        ->getNamedObjectsDictionary(pNamedobj, AcDb::kForRead);

    // Get a pointer to the ASDK_DICT dictionary.
    //
    AcDbDictionary *pDict;
    pNamedobj->getAt("ASDK_DICT", (AcDbObject*&)pDict,
        AcDb::kForRead);
    pNamedobj->close();

    // Get an iterator for the ASDK_DICT dictionary.
    //
    AcDbDictionaryIterator* pDictIter= pDict->newIterator();
    AsdkMyClass *pMyCl;
    Adesk::Int16 val;
    for (; !pDictIter->done(); pDictIter->next()) {

        // Get the current record, open it for read, and
        // print its data.
        //
        pDictIter->getObject((AcDbObject*&)pMyCl,
            AcDb::kForRead);
        pMyCl->getData(val);
        pMyCl->close();
        acutPrintf("\nintval is:  %d", val);
    }
    delete pDictIter;
    pDict->close();
}

// The initialization function called from the acrxEntryPoint()
// function during the kInitAppMsg case is used to add commands
// to the command stack and to add classes to the ACRX class
// hierarchy.
//
void
initApp()
{
    acedRegCmds->addCommand("ASDK_DICTIONARY_COMMANDS",
        "ASDK_CREATE", "CREATE", ACRX_CMD_MODAL,
        createDictionary);
    acedRegCmds->addCommand("ASDK_DICTIONARY_COMMANDS",
        "ASDK_ITERATE", "ITERATE", ACRX_CMD_MODAL,
        iterateDictionary);
    AsdkMyClass::rxInit();
    acrxBuildClassHierarchy();
}
```

```
// The cleanup function called from the acrxEntryPoint() function
// during the kUnloadAppMsg case removes this application's
// command set from the command stack and removes this application's
// custom classes from the ACRX runtime class hierarchy.
//
void
unloadApp()
{
    acedRegCmds->removeGroup("ASDK_DICTIONARY_COMMANDS");

    // Remove the AsdkMyClass class from the ACRX runtime
    // class hierarchy.  If this is done while the database is
    // still active, it should cause all objects of class
    // AsdkMyClass to be turned into proxies.
    //
    deleteAcRxClass(AsdkMyClass::desc());
}
```

Object Version Support

Several mechanisms were formerly used for managing versions of custom object classes used in ObjectARX applications:

- Renaming the class for each new version.
- Maintaining a version number as the first data member of the class.
- Maintaining the version number as extended data (`xdata`) or in an extension dictionary.

These mechanisms have been superseded by the class versioning system. These earlier mechanisms are described below (following the description of class versioning), to help maintain code that uses them.

Class Versioning

Beginning with AutoCAD 2000, every custom class must provide a drawing and maintenance version number. The drawing value corresponds to the release of AutoCAD that was current when the class was created. The maintenance value can be set to whatever is appropriate for your class. For ObjectARX classes, the maintenance value will be set to zero every time the drawing version changes due to a new AutoCAD release. The version values are defined in the *acdb.h* header file. The `ACRX_DXF_DEFINE_MEMBERS` macro has been modified in AutoCAD 2000 to take two new arguments, `DWG_VERSION` and `MAINTENANCE_VERSION`:

```
#define ACRX_DXF_DEFINE_MEMBERS(CLASS_NAME,PARENT_CLASS,\
    DWG_VERSION,MAINTENANCE_VERSION,PROXY_FLAGS,DXF_NAME,APP)
```

These two arguments must be provided, and there are no defaults. The new arguments specify the version when the class was introduced. They become data members of the `AcRxClass`, but they are not persistent, that is, they are not stored in the class section of DWG and DXF files.

Class Versioning Example

The new `ACRX_DXF_DEFINE_MEMBERS` arguments determine which version to use when an object is to file itself out. When a drawing is saved, you can determine which DWG version to save to by calling the `dwgVersion()` method of the filer. It is *not* necessary that the object file itself out as the same version as the filer. Previous releases of AutoCAD have done exactly that, which has led to problems that are best described by the following example.

In Release 14, a new data member (`mTreatElementsAsHard`) was added to `AcDbDictionary`. In AutoCAD 2000, a class called `AcDbDictionaryWithDefault` has been derived from `AcDbDictionary`. When using `acdbSaveAsR13()`, the `mTreatElementsAsHard` member is not written out since Release 13 doesn't know about the member. If the drawing saved by `acdbSavedAsR13()` is next opened by Release 14, the instance of `AcDbDictionaryWithDefault` becomes a proxy, since Release 14 doesn't include this class. When an `AcDbObject` becomes a proxy, all the data below the `AcDbObject` level is kept intact by AutoCAD as "proxy data" and is not changed. When the drawing is saved by Release 14, the data is dumped back to the DWG file as it was read in. The result is a Release 14 DWG file that has an instance of `AcDbDictionaryWithDefault`, but is missing the `mTreatElementsAsHard` data. When reading in this drawing with AutoCAD 2000, AutoCAD (specifically, `AcDbDictionary::dwgInFields()`) looks for that data member since it recognizes the filer being a Release 14 type that should have the `mTreatElementsAsHard` data. However, the data is not present, the sequence is lost, and the drawing is corrupt.

This is not specific to `AcDbDictionaryWithDefault`. New classes in AutoCAD 2000, already introduced by ObjectARX or that will be introduced by third parties, can suffer from this problem, especially if one of their superclasses has changed data.

Using Class Versioning

In the example above, the `AcDbDictionaryWithDefault` object should have been filed out with the AutoCAD 2000 version of its data since it becomes a proxy in all previous versions and no one will be reading its data (except for the data filed out at the `AcDbObject` level).

To fix this, a mechanism has been introduced where the object can override the filer version and dictate what version it wants to be filed out or in with. The following rules apply:

1. If the filer version is older than the version of AutoCAD that the object first appeared in (the "birth" version), use the object's birth version.
2. If the filer version is the same or newer than the birth version of the object, use the filer version.

The appropriate rule should be used by the leaf class, as well as all its base classes, to file data in and out. In the example given above, rule 2 applies (the filer is from AutoCAD 2000, while the object is from Release 14), so we file out using the AutoCAD 2000 version. If there was a new class introduced in Release 14 whose data is also changing in AutoCAD 2000, and the operation is to save as Release 13, rule 1 applies and we file out using the Release 14 (birth) version.

Two new virtual methods of `AcDbObject` have been introduced to implement class versioning, one for DWG and one for DXF files:

```
virtual Acad::ErrorStatus
getObjectSaveVersion(
    const AcDbDwgFiler* pFiler,
    AcDb::AcDbDwgVersion& ver,
    AcDb::MaintenanceReleaseVersion& maintVer);

virtual Acad::ErrorStatus
getObjectSaveVersion(
    const AcDbDxfFiler* pFiler,
    AcDb::AcDbDwgVersion& ver,
    AcDb::MaintenanceReleaseVersion& maintVer);
```

In the filer methods, instead of calling `filer->dwgVersion()`, call `self()->getObjectSaveVersion(filer, ...)` to let the object indicate which version to use to dump the data out. Similarly, call that method in `dwgInFields()` and `dxfInFields()` to find out which version the data is coming back in.

Since not all the objects have a need to override the filer version, the ones that do need to do so specify their intent by setting a bit on the object. This would normally be done in the constructor of the class. The bit is used as a quick check to determine if it's necessary to override the filer version. Methods related to this have been added to `AcDbObject`:

```
bool
hasSaveVersionOverride();

void
setHasSaveVersionOverride(
    bool bSetIt);
```

There is also a new `AcDbObject` method to get the birth version of the object:

```
Acad::ErrorStatus
getObjectBirthVersion(
    AcDb::AcDbDwgVersion& ver,
    AcDb::MaintenanceReleaseVersion& maintVer);
```

This method returns the two version numbers stored with the `AcRxClass` of this object, which are specified while registering the class using the `ACRX_DXF_DEFINE_MEMBERS` macro.

Implementing Class Versioning

1. If you are deriving a class from any ObjectARX classes, except for `AcDbObject` and `AcDbEntity`, call `setHasSaveVersionOverride(true)` in the constructor so that the `AcDbObject::getObjectSaveVersion()` default implementation knows to not just return the filer version, but to instead check with your class version and return an appropriate "object save version" according to the rules described above. `getObjectSaveVersion()` doesn't do that unless this bit is set.
2. You can override `AcDbObject::getObjectSaveVersion()` to specify which version the object data needs to be stored in. There is no need to supermessage because you are completely taking over.
3. Do *not* use `filer->dwgVersion()` in your `dwgInFields()`, `dwgOutFields()`, `dxfInFields()`, or `dxfOutFields()` methods. Use `self()->getObjectSaveVersion()` instead. Its default implementation is to return `filer->dwgVersion()` unless the object wants to override the save version.

   If you use `filer->dwgVersion()`, you're disabling proper filer selection for the classes derived from yours.
4. Be sure to register your classes using `ACRX_DXF_DEFINE_MEMBERS` in AutoCAD 2000 with a "birth version" using the two new arguments. Remember that birth version means the version of AutoCAD that the class was introduced in, and that will not always be AutoCAD 2000, but could be Release 13 or Release 14.

Class Renaming

Renaming classes for each new version is the simplest method, as it does not involve implementing new data elements or functions to detect and respond to different class version numbers.

Class Data or Xdata Version Numbers

The version number can be stored as an 8-bit integer data member (of type `Adesk::UInt8`) of the class, and can be filed in and out as the first data member for each object. Because this data is persistent, and is the first item read, it can be checked to determine the version of the object before any other data is read.

When a number is used to differentiate versions of an object, the parent ObjectARX application must be able to handle these two cases of incompatible versions of objects:

- When the application encounters an outdated version of an object in a file, it should be able to update the object to the current version. Updating an old object involves adding any new data members and member functions, as well as changing the version number.
- When an older version of the application encounters a newer version of an object (that is, when the revision number of an object is greater than the revision number of the application), the custom class's `dxfInFields()` and `dwgInFields()` functions should immediately return the error code `eMakeMeProxy` to AutoCAD. AutoCAD will then create a proxy object for the drawing session, and write the original object to file when the drawing is saved.

Object versioning with a data-member version number is illustrated in the following code fragments from *\objectarx\samples\polysamp\poly.cpp* in the ObjectARX SDK.

```
// Object Version
#define VERSION 1
...

Acad::ErrorStatus
AsdkPoly::dwgInFields(AcDbDwgFiler* filer)
{
...
    // Object Version - must always be the first item
    Adesk::Int16 version;
    filer->readItem(&version);
    if (version > VERSION)
        return Acad::eMakeMeProxy;
...
}
```

```
Acad::ErrorStatus
AsdkPoly::dwgOutFields(AcDbDwgFiler* filer) const
{
...
    // Object Version - must always be the first item
    Adesk::Int16 version = VERSION;
    filer->writeItem(version);
...
}


Acad::ErrorStatus
AsdkPoly::dxfInFields(AcDbDxfFiler* filer)
{
...
    // Object Version
        case AcDb::kDxfInt16:
            Adesk::Int16 version;
            version = rb.resval.rint;
            if (version > VERSION)
                return Acad::eMakeMeProxy;
            break;
...
}


Acad::ErrorStatus
AsdkPoly::dxfOutFields(AcDbDxfFiler* filer) const
{
...
    // Object Version
    Adesk::Int16 version = VERSION;
    filer->writeItem(AcDb::kDxfInt16, version);
...
}
```

Deriving from AcDbEntity

13

In This Chapter

- Deriving Custom Entities
- Overriding Common Entity Functions
- Extending Entity Functionality
- Using AcEdJig

This chapter describes how to derive a custom class from `AcDbEntity`, and includes specific examples of overriding virtual methods provided by the `AcDbEntity` class. Overriding common entity operations, such as object snap points, grip points, and stretch points, is also discussed in this chapter.

The material in this chapter assumes you are familiar with the material presented in chapter 6, “Entities”; chapter 11, “Deriving a Custom ObjectARX Class”; and chapter 12, “Deriving from AcDbObject.”

Deriving Custom Entities

`AcDbEntity` is the base class for all database objects having a graphical representation. `AcDbEntity` is derived from `AcDbObject`. Creating a custom entity involves the following steps.

To create a custom entity

1. Derive a custom class from `AcDbEntity`.
2. Override all of the necessary `AcDbObject` functions. See chapter 12, "Deriving from AcDbObject."
3. Override the required `AcDbEntity` functions. This will be discussed in the following sections.
4. Override other functions as needed to support your custom functionality.
5. If you want to support the MATCHPROP command, implement `AcDbMatchProperties` as a protocol extension.
6. If you want to create a custom drag sequence for your entity, implement your own version of `AcEdJig`.

The following sections discuss these topics in more detail.

AcDbEntity Functions to Override

The following functions must be overridden when you derive a custom class from `AcDbEntity`:

```
virtual Adesk::Boolean
worldDraw(
    AcGiWorldDraw* mode);

virtual Acad::ErrorStatus
getGeomExtents(
    AcDbExtents& extents) const;

virtual Acad::ErrorStatus
transformBy(
    const AcGeMatrix3d& xform);

virtual Acad::ErrorStatus
getTransformedCopy(
    const AcGeMatrix3d& xform,
    AcDbEntity*& ent) const;
```

```
virtual Acad::ErrorStatus
getGripPoints(
    AcGePoint3dArray& gripPoints,
    AcDbIntArray&  osnapModes,
    AcDbIntArray&  geomIds) const;

virtual Acad::ErrorStatus
moveGripPointsAt(
    const AcDbIntArray& indices,
    const AcGeVector3d& offset);
```

AcDbEntity Functions Usually Overridden

The following functions are usually overridden when deriving a custom class from `AcDbEntity`. Whether or not you override these functions depends on the custom functionality that your class supports.

```
virtual void
viewportDraw(AcGiViewportDraw* mode);

virtual void
list() const;

virtual Acad::ErrorStatus
intersectWith(
    const AcDbEntity* ent,
    AcDb::Intersect intType,
    AcGePoint3dArray& points,
    int thisGsMarker = 0,
    int otherGsMarker = 0) const;

virtual Acad::ErrorStatus
intersectWith(
    const AcDbEntity* ent,
    AcDb::Intersect intType,
    const AcGePlane& projPlane,
    AcGePoint3dArray& points,
    int thisGsMarker = 0,
    int otherGsMarker = 0) const;

virtual Acad::ErrorStatus
getOsnapPoints(
    AcDb::OsnapMode osnapMode,
    int gsSelectionMark,
    const AcGePoint3d& pickPoint,
    const AcGePoint3d& lastPoint,
    const AcGeMatrix3d& viewXform,
    AcGePoint3dArray& snapPoints,
    AcDbIntArray& geomIds) const;

virtual Acad::ErrorStatus
getStretchPoints(
    AcGePoint3dArray&) const;
```

```
virtual Acad::ErrorStatus
moveStretchPointsAt(
    const AcDbIntArray& indices,
    const AcGeVector3d& offset);

virtual Acad::ErrorStatus
explode(
    AcDbVoidPtrArray& entitySet) const;


virtual Acad::ErrorStatus
getSubentPathsAtGsMarker(
    AcDb::SubentType type,
    int gsMark,
    const AcGePoint3d& pickPoint,
    const AcGeMatrix3d& viewXform,
    int& numPaths,
    AcDbFullSubentPath* & subentPaths,
    int numInserts = 0,
    AcDbObjectId* entAndInsertStack = NULL) const;

virtual Acad::ErrorStatus
applyPartialUndo(
    AcDbDwgFiler* undoFiler,
    AcRxClass* classObj);

virtual void
subSetDatabaseDefaults(
    AcDbDatabase* pDb);

virtual void
saveAs(
    AcGiWorldDraw* mode, AcDb::SaveType st);
```

AcDbEntity Functions Rarely Overridden

The following `AcDbEntity` functions are rarely overridden:

```
virtual Acad::ErrorStatus
setColor(
    const AcCmColor &color);

virtual Acad::ErrorStatus
setColorIndex(
    Adesk::UInt16 color);

virtual Acad::ErrorStatus
setLinetype(
    const char* newVal);

virtual Acad::ErrorStatus
setLinetype(
    AcDbObjectId newVal);
```

```
virtual  void
getEcs(
    AcGeMatrix3d& retVal) const;

virtual  Acad::ErrorStatus
getGsMarkersAtSubentPath(
    const AcDbFullSubentPath& subPath,
    AcDbIntArray& gsMarkers) const;

virtual Acad::ErrorStatus
highlight(
    const AcDbFullSubentPath& subId = kNullSubent) const;

virtual Acad::ErrorStatus
unhighlight(
    const AcDbFullSubentPath& subId = kNullSubent) const;

virtual AcDbEntity*
subentPtr(

    const AcDbFullSubentPath& id) const;

virtual Adesk::Boolean
saveImagesByDefault()  const;

virtual void
setAttributes(
    AcGiSubEntityTraits* pTraits);
```

The following sections discuss overriding several commonly used functions.

Overriding Common Entity Functions

Common entity functions are described in chapter 6, "Entities." This chapter assumes you are familiar with the material presented there. The following sections describe how to override functions that display entities and functions that use object snap points, grip points, and stretch points. In addition, overriding transformation, intersection, and explode functions are discussed.

Overriding worldDraw() and viewportDraw()

AutoCAD calls the `worldDraw()` and `viewportDraw()` functions to display the entity. You must implement the `worldDraw()` function for any class derived from `AcDbEntity`. The `viewportDraw()` function is optional.

```
virtual Adesk::Boolean
AcDbEntity::worldDraw(
    AcGiWorldDraw *pWd);
```

```
virtual void
AcDbEntity::viewportDraw(
    AcGiViewportDraw *pVd);
```

Whenever AutoCAD needs to regenerate the graphics to display an entity, the `worldDraw()` and `viewportDraw()` functions are called in the following manner:

```
if (!entity->worldDraw(pWd))
    for (each relevant viewport)
        entity->viewportDraw(pVd);
```

The `worldDraw()` function builds the portion of the entity's graphical representation that can be specified independently of any particular model-space view or paper-space viewport contexts. The `viewportDraw()` function then builds the view-dependent portion of the entity's graphics. If any of the entity's graphics are view-dependent, the `worldDraw()` function must return `kFalse` and the `viewportDraw()` function must be implemented. Conversely, if the entity has no view-dependent graphics, then the `worldDraw()` function must return `kTrue, and` the custom entity does not implement the `viewportDraw()` function.

The `AcDbEntity::worldDraw()` function takes a pointer to an `AcGiWorldDraw` object. `AcGiWorldDraw` is a container class for the AcGi geometry and traits objects. Specifically, `AcGiWorldDraw` contains two other objects:

- `AcGiWorldGeometry`
- `AcGiSubEntityTraits`

The `AcGiWorldGeometry` object can be accessed from within the `worldDraw()` function by using the `AcGiWorldDraw::geometry()` function, and the `AcGiSubEntityTraits` object can be accessed by using the `AcGiWorldDraw::subEntityTraits()` function.

The `AcGiWorldGeometry` object writes vectors to AutoCAD's refresh memory using its set of drawing primitives. A *primitive* is the lowest-level instruction used to draw graphical entities. The world geometry object has the following functions for drawing primitives in world coordinates:

- Circle
- Circular arc
- Polyline
- Polygon
- Mesh
- Shell
- Text
- Xline
- Ray

The `AcGiSubEntityTraits` object sets graphical attribute values using its set of traits functions:

- Color
- Layer
- Linetype
- Polygon fill type
- Selection marker

The `AcDbEntity::viewportDraw()` function takes a pointer to an `AcGiViewportDraw` object and builds the view-specific representation of an entity. The viewport draw object is also a container object for other objects, which include the following:

- `AcGiViewportGeometry`
- `AcGiSubEntityTraits`
- `AcGiViewport`

The viewport geometry object provides the same list of primitives as the world geometry object and adds to it the following primitives, which use eye- and display-space coordinates to draw polylines and polygons:

- `polylineEye()`
- `polygonEye()`
- `polylineDc()`
- `polygonDc()`

The viewport subentity traits object is the same as that used by the world draw object (`AcGiSubEntityTraits`). The viewport object provides functions for querying the viewport's transformation matrices and viewing parameters.

WARNING! An AcGi object such as `AcGiWorldDraw` or `AcGiViewportDraw` should not be stored as a global or static variable. Do not save copies of AcGi objects across calls to the `worldDraw()` and `viewportDraw()` functions. Once these functions return, the AcGi objects are no longer valid.

For more information about the AcGi library, see chapter 26, "The Graphics Interface Library."

Overriding saveAs()

You should override `saveAs()` if you want to save an alternate graphical representation for saving proxy entity graphics, Release 12 DWG files, or both. If your custom entity doesn't override the `AcDbEntity::saveAs()` function, AutoCAD will leverage your `worldDraw()` function to support proxy entity

graphics or Release 12 DWG files. The `AcDbEntity::saveAs()` function merely calls the `worldDraw()` function.

```
virtual void
AcDbEntity::saveAs(
    AcGiWorldDraw *pWd,
    AcDb::SaveType saveType);
```

The `saveType` parameter is used when you want to build unique, alternate graphical representations for both kinds of saving; it indicates for which purpose `saveAs()` was called. The `saveType` parameter has either of the following values:

- `kR13Save` indicates that `saveAs()` was called to save proxy graphics data.
- `kR12Save` indicates that `saveAs()` was called for saving to Release 12 DWG files.

From within `saveAs()`, you may want to call the `worldDraw()` function for one value of `saveType` and make direct `AcGiWorldGeometry` and `AcGiSubEntityTraits` calls for the other value, or you may not want to call the `worldDraw()` function at all.

In either case, before calling `saveAs()`, AutoCAD first replaces `AcGiWorldDraw`'s geometry and traits objects with special subclasses of `AcGiWorldGeometry` and `AcGiSubEntityTraits`. These subclasses's geometric primitive and property traits functions cache the data in the appropriate format rather than performing a display. After calling `saveAs()`, AutoCAD writes the cached data to disk.

Neither kind of saving permits preserving any view-dependent graphics. The `viewportDraw()` function is not called as part of either of the save operations. Your custom entity may rely on its `viewportDraw()` function for its graphics, so its `worldDraw()` function alone would not produce an appropriate image. In that case, you'll need to override `saveAs()` to produce reasonable graphics for Release 12 and proxy objects.

For more information on proxy graphics data, see chapter 14, "Proxy Objects."

In Release 12 DWG files, information about the original entity is not saved in the file. However, the first Release 12 entity will have the same handle as the original entity, and any additional Release 12 entities will have the original entity handle placed in their xdata. (Look under the application name `ACAD`, following the string data member `R13OBJECT`.) This feature is provided so that you can group the Release 12 entities into a block.

Implementing the Object Snap Point Function

You'll need to override the `getOsnapPoints()` function if you want your custom entity to support object snap modes. AutoCAD invokes this function to acquire the relevant snap points for the current mode. If you do not want your entity to support snap points for a particular mode, you can filter out the snap modes you support and return `eOk` for the others; AutoCAD will prompt the user to select again. If multiple object snap modes are active, this function is called once for each object snap mode.

NOTE The intersection object snap mode is processed differently from `getOsnapPoints()`. It uses `AcDbEntity::intersectWith()`, not `getOsnapPoints()`.

The following shows how the `AsdkPoly` class implements the `getOsnapPoints()` function:

```
Acad::ErrorStatus
AsdkPoly::getOsnapPoints(
    AcDb::OsnapMode       osnapMode,
    int                   gsSelectionMark,
    const AcGePoint3d&    pickPoint,
    const AcGePoint3d&    lastPoint,
    const AcGeMatrix3d&   viewXform,
    AcGePoint3dArray&     snapPoints,
    AcDbIntArray&         /*geomIds*/) const
{
    assertReadEnabled();
    Acad::ErrorStatus es = Acad::eOk;
    if (gsSelectionMark == 0)
        return Acad::eOk;
    if (   osnapMode != AcDb::kOsModeEnd
        && osnapMode != AcDb::kOsModeMid
        && osnapMode != AcDb::kOsModeNear
        && osnapMode != AcDb::kOsModePerp
        && osnapMode != AcDb::kOsModeCen
        && osnapMode != AcDb::kOsModeIns)
    {
        return Acad::eOk;
    }

    // First, check to see if the gsSelection marker is the
    // text geometry. If so, handle center and insertion
    // modes, then return. No need to go into perp, mid, etc.
    //
    AcGePoint3d center;
    getCenter(center);
```

```
if (gsSelectionMark == (mNumSides + 1)) {
    if (osnapMode == AcDb::kOsModeIns)
            snapPoints.append(center);
        else if (osnapMode == AcDb::kOsModeCen)
            snapPoints.append(center);

        return es;
    }

int startIndex = gsSelectionMark - 1;
AcGePoint3dArray vertexArray;
if ((es = getVertices3d(vertexArray)) != Acad::eOk) {
    return es;
}

AcGeLineSeg3d lnsg(vertexArray[startIndex],
        vertexArray[startIndex + 1]);
AcGePoint3d pt;
AcGeLine3d line, perpLine;
AcGeVector3d vec;
AcGeVector3d viewDir(viewXform(Z, 0), viewXform(Z, 1),
        viewXform(Z, 2));

switch (osnapMode) {
case AcDb::kOsModeEnd:
    snapPoints.append(vertexArray[startIndex]);
    snapPoints.append(vertexArray[startIndex + 1]);
    break;
case AcDb::kOsModeMid:
    pt.set(
        ((vertexArray[startIndex])[X]
            + (vertexArray[startIndex + 1])[X]) * 0.5,
        ((vertexArray[startIndex])[Y]
            + (vertexArray[startIndex + 1])[Y]) * 0.5,
        ((vertexArray[startIndex])[Z]
            + (vertexArray[startIndex + 1])[Z]) * 0.5);
    snapPoints.append(pt);
    break;
case AcDb::kOsModeNear:
    pt = lnsg.projClosestPointTo(pickPoint, viewDir);
    snapPoints.append(pt);
    break;
case AcDb::kOsModePerp:
    // Create a semi-infinite line and find a point on it.
    //
    vec = vertexArray[startIndex + 1]
        - vertexArray[startIndex];
    vec.normalize();
    line.set(vertexArray[startIndex], vec);
    pt = line.closestPointTo(lastPoint);
    snapPoints.append(pt);
    break;
```

```
    case AcDb::kOsModeCen:
        snapPoints.append(center);
        break;
    default:
        return Acad::eOk;
    }
    return es;
}
```

Implementing the Grip Point Functions

AutoCAD entities have grip points that appear when the user selects an entity with the pointing device. The `getGripPoints()` function returns the grip points that have been defined for an entity.

The signatures for the `getGripPoints()` and `moveGripPointsAt()` functions for `AcDbEntity` are

```
virtual Acad::ErrorStatus
AcDbEntity::getGripPoints(
    AcGePoint3dArray&  gripPoints,
    AcDbIntArray&      osnapModes,
    AcDbIntArray&      geomIds) const;

virtual Acad::ErrorStatus
AcDbEntity::moveGripPointsAt(
    const AcDbIntArray&  indices,
    const AcGeVector3d&  offset);
```

The `osnapModes` and `geomIds` arguments of the `getGripPoints()` function are not currently used.

Stretch mode in grip editing allows you to stretch an object by moving selected grips to new locations. AutoCAD calls the `moveGripPointsAt()` function when the user is in stretch mode. For certain entities, however, some grips move the object rather than stretching it. These grips include grips on text objects, blocks, midpoints of lines, centers of circles, centers of ellipses, and point objects. In these cases, the `moveGripPointsAt()` function calls `transformBy()`.

NOTE The default implementation of the `AcDbEntity::moveGripPointsAt()` function is to invoke the `transformBy()` function.

When the user is in grip move, rotate, scale, or mirror modes, AutoCAD calls the `transformBy()` function, described in chapter 6, “Entities.”

If you want the user to be able to edit your entity using grips, you'll need to override the `getGripPoints()` and `moveGripPointsAt()` functions. The entity defines its grip points and how to interpret the user-supplied offset.

The following excerpt shows how the custom `AsdkPoly` class implements these functions. The object defined by this class has a grip point at each vertex and a grip point at its center. These grip points are returned by the `getGripPoints()` function. If the user selects a grip point when in grip stretch mode, AutoCAD invokes the `moveGripPointsAt()` function passing in an array of the indexes for the selected grip points and a 3D vector specifying how much the user moved the pointing device. If the user has selected a vertex grip point, the polygon is stretched uniformly by the specified offset. If the user picked the center grip point, the polygon is simply translated by an amount equal to the offset (this value is passed to the `transformBy()` function, as shown here).

```
Acad::ErrorStatus
AsdkPoly::getGripPoints(
    AcGePoint3dArray& gripPoints,
    AcDbIntArray& osnapModes,
    AcDbIntArray& geomIds) const
{
    assertReadEnabled();
    Acad::ErrorStatus es;
    if ((es = getVertices3d(gripPoints)) != Acad::eOk) {
        return es;
    }

    // Remove the duplicate point at the start/end and add
    // center as the last point.
    //
    gripPoints.removeAt(gripPoints.length() - 1);
    AcGePoint3d center;
    getCenter(center);
    gripPoints.append(center);
    return es;
}

Acad::ErrorStatus
AsdkPoly::moveGripPointsAt(
    const AcDbIntArray& indices,
    const AcGeVector3d& offset)
{
    if (indices.length()== 0 || offset.isZeroLength())
        return Acad::eOk; //that's easy :-)
    if (mDragDataFlags & kCloneMeForDraggingCalled) {
        mDragDataFlags &= kUseDragCache;
```

```
        // We need to make sure that all the poly's drag data members
         // are in sync with the true data members.
         //
         //mDragCenter = mCenter;
         //mDragStartPoint = mStartPoint;
    } else

         // Only if we're not dragging do we want to make an undo
         // recording and check if the object's open for write.
         //
         assertWriteEnabled();

    //if there's more than one hot vertex or there's one and it is
    //the center then simply transform.
    if (indices.length()>1 || indices[0] == mNumSides)
         return transformBy(AcGeMatrix3d::translation(offset));

    AcGeVector3d off(offset);

    // Calculate the offset vector of the startpoint
    // from the offset vector on a vertex.
    double rotateBy = 2.0 * 3.14159265358979323846 /
         mNumSides * indices[0];
    AcGePoint3d cent;
    getCenter(cent);
    off.transformBy(AcGeMatrix3d::rotation(rotateBy,
         normal(),cent));
    acdbWcs2Ecs(asDblArray(off),asDblArray(off),
         asDblArray(normal()),Adesk::kTrue);

    if (mDragDataFlags & kUseDragCache){
         mDragStartPoint = mStartPoint + AcGeVector2d(off.x,off.y);
         mDragElevation = mElevation + off.z;
    } else{
         mStartPoint = mStartPoint + AcGeVector2d(off.x,off.y);
         mElevation = mElevation + off.z;
    }
    return Acad::eOk;
}
```

Implementing the Stretch Point Functions

The set of stretch points for an entity is often a subset of its grip points. When the user invokes the STRETCH command, the `getStretchPoints()` function is used to return the stretch points defined for the selected entity. For many entities, grip mode and stretch mode are identical. The implementation for the `AcDbEntity::getStretchPoints()` function and the `AcDbEntity::moveStretchPointsAt()` function is to invoke your `getGripPoints()` and `moveGripPointsAt()` functions.

The signatures for the stretch functions are

```
virtual Acad::ErrorStatus
AcDbEntity::getStretchPoints(
    AcGePoint3dArray& stretchPoints) const;

virtual Acad::ErrorStatus
AcDbEntity::moveStretchPointsAt(
    const AcDbIntArray&  indices,
    const AcGeVector3d&  offset);
```

You are not required to override the `getStretchPoints()` and `moveStretchPointsAt()` functions of `AcDbEntity`, because they default to the `getGripPoints()` and `transformBy()` functions.

The custom `AsdkPoly` class overrides these functions as shown in the example in this section. The `getStretchPoints()` function returns the vertices of the polygon, but not the center. The `moveStretchPointsAt()` function checks whether all the stretch points have been selected. If they have, it invokes the `transformBy()` function. Otherwise, it invokes the `moveGripPointsAt()` function.

```
Acad::ErrorStatus
AsdkPoly::getStretchPoints(
    AcGePoint3dArray& stretchPoints) const
{
    assertReadEnabled();
    Acad::ErrorStatus es;
    if ((es = getVertices3d(stretchPoints))
        != Acad::eOk)
    {
        return es;
    }
    // Remove the duplicate point at the start and end.
    //
    stretchPoints.removeAt(stretchPoints.length() - 1);
    return es;
}

Acad::ErrorStatus
AsdkPoly::moveStretchPointsAt(
    const AcDbIntArray& indices,
    const AcGeVector3d& offset)
{
    return moveGripPointsAt(indices, offset);
}
```

Transformation Functions

The `AcDbEntity` class offers two transformation functions. The `transformBy()` function applies a matrix to an entity. The `getTransformedCopy()` function enables an entity to return a copy of itself with the transformation applied to it.

If an entity is uniformly scaled and orthogonal, the default implementation of the `AcDbEntity::getTransformedCopy()` function clones the entity and then invokes the `transformBy()` function on the cloned entity. (Use the `AcGeMatrix3d::isUniScaledOrtho()` function to determine if the input matrix is uniformly scaled and orthogonal.)

The custom `AsdkPoly` class overrides both the `transformBy()` function and the `getTransformedCopy()` function. When `AsdkPoly` is nonuniformly scaled, it becomes a polyline.

```
Acad::ErrorStatus
AsdkPoly::transformBy(const AcGeMatrix3d& xform)
{
    // If we're dragging, we aren't really going to change our
    // data, so we don't want to make an undo recording nor do
    // we really care if the object's open for write.
    //
    if (mDragDataFlags & kCloneMeForDraggingCalled) {
        mDragDataFlags &= kUseDragCache;
        mDragPlaneNormal = mPlaneNormal;
        mDragElevation = mElevation;

       AcGeMatrix2d xform2d(xform.convertToLocal(mDragPlaneNormal,
            mDragElevation));
        mDragCenter = xform2d * center();
        mDragStartPoint = xform2d * startPoint();
        mDragPlaneNormal.normalize();

    } else {
        assertWriteEnabled();

        AcGeMatrix2d xform2d(xform.convertToLocal(mPlaneNormal,
            mElevation));
        mCenter.transformBy(xform2d);
        mStartPoint.transformBy(xform2d);
        mPlaneNormal.normalize();
    }
    return Acad::eOk;
}
```

```
Acad::ErrorStatus AsdkPoly::getTransformedCopy(
    const AcGeMatrix3d& mat,
    AcDbEntity*& ent) const
{
    assertReadEnabled();
    Acad::ErrorStatus es = Acad::eOk;
    AcGePoint3dArray vertexArray;
    if ((es = getVertices3d(vertexArray))
        != Acad::eOk)
    {
        return es;
    }

    for (int i = 0; i < vertexArray.length(); i++) {
        vertexArray[i].transformBy(mat);
    }

    AcDbSpline *pSpline = NULL;

    if ((es = rx_makeSpline(vertexArray, pSpline))
        != Acad::eOk)
    {
        return es;
    }

    assert(pSpline != NULL);
    pSpline->setPropertiesFrom(this);
    ent = pSpline;
    return es;
}
```

Intersecting with Other Entities

The `intersectWith()` function has two forms:

```
virtual  Acad::ErrorStatus
intersectWith(
    const AcDbEntity* ent,
    AcDb::Intersect intType,
    AcGePoint3dArray& points,
    int thisGsMarker = 0,
    int otherGsMarker = 0) const;

virtual  Acad::ErrorStatus
intersectWith(
    const AcDbEntity* ent,
    AcDb::Intersect intType,
    const AcGePlane& projPlane,
    AcGePoint3dArray& points,
    int thisGsMarker = 0,
    int otherGsMarker = 0) const;
```

The first form of the `intersectWith()` function tests for simple intersection of two entities. The second form calculates the intersection on a projection plane. However, both functions return the intersection points on the entity itself.

To use the projection plane form of the intersectWith() function

1. Project your entity and the argument entity onto the plane.
2. Test the entities for intersection on the projection plane.
3. Project the intersection points back onto the entity and return them.

The custom `AsdkPoly` class overrides both forms of the `intersectWith()` function.

```
Acad::ErrorStatus
AsdkPoly::intersectWith(
    const AcDbEntity* ent,
    AcDb::Intersect intType,
    AcGePoint3dArray& points,
    int /*thisGsMarker*/,
    int /*otherGsMarker*/) const
{
    assertReadEnabled();
    Acad::ErrorStatus es = Acad::eOk;
    if (ent == NULL)
        return Acad::eNullEntityPointer;

    // The idea is to intersect each side of the polygon
    // with the given entity and return all the points.
    //
    // For non-R12-entities with intersection methods defined,
    // we call that method for each of the sides of the polygon.
    // For R12-entities, we use the locally defined intersectors,
    // since their protocols are not implemented.
    //
    if (ent->isKindOf(AcDbLine::desc())) {
        if ((es = intLine(this, AcDbLine::cast(ent),
            intType, NULL, points)) != Acad::eOk)
        {
            return es;
        }
    } else if (ent->isKindOf(AcDbArc::desc())) {
        if ((es = intArc(this, AcDbArc::cast(ent), intType,
            NULL, points)) != Acad::eOk)
        {
            return es;
        }
    } else if (ent->isKindOf(AcDbCircle::desc())) {
        if ((es = intCircle(this, AcDbCircle::cast(ent),
            intType, NULL, points)) != Acad::eOk)
        {
            return es;
        }
```

```
    } else if (ent->isKindOf(AcDb2dPolyline::desc())) {
        if ((es = intPline(this, AcDb2dPolyline::cast(ent),
            intType, NULL, points)) != Acad::eOk)
        {
            return es;
        }
    } else if (ent->isKindOf(AcDb3dPolyline::desc())) {
        if ((es = intPline(this, AcDb3dPolyline::cast(ent),
            intType, NULL, points)) != Acad::eOk)
        {
            return es;
        }
    } else {
        AcGePoint3dArray vertexArray;
        if ((es = getVertices3d(vertexArray))
            != Acad::eOk)
        {
            return es;
        }
        if (intType == AcDb::kExtendArg
            || intType == AcDb::kExtendBoth)
        {
            intType = AcDb::kExtendThis;
        }

        AcDbLine *pAcadLine;

        for (int i = 0; i < vertexArray.length() - 1; i++) {
            pAcadLine = new AcDbLine();
            pAcadLine->setStartPoint(vertexArray[i]);
            pAcadLine->setEndPoint(vertexArray[i + 1]);
            pAcadLine->setNormal(normal());

            if ((es = ent->intersectWith(pAcadLine, intType,
                points)) != Acad::eOk)
            {
                delete pAcadLine;
                return es;
            }
            delete pAcadLine;
        }
    }
    return es;
}

Acad::ErrorStatus
AsdkPoly::intersectWith(
    const AcDbEntity* ent,
    AcDb::Intersect intType,
    const AcGePlane& projPlane,
    AcGePoint3dArray& points,
    int /*thisGsMarker*/,
    int /*otherGsMarker*/) const
```

```
{
    assertReadEnabled();
    Acad::ErrorStatus es = Acad::eOk;
    if (ent == NULL)
        return Acad::eNullEntityPointer;

    // The idea is to intersect each side of the polygon
    // with the given entity and return all the points.
    //
    // For non-R12-entities, with intersection methods defined,
    // we call that method for each of the sides of the polygon.
    // For R12-entities, we use the locally defined intersectors,
    // since their protocols are not implemented.
    //
    if (ent->isKindOf(AcDbLine::desc())) {
        if ((es = intLine(this, AcDbLine::cast(ent),
            intType, &projPlane, points)) != Acad::eOk)
        {
            return es;
        }
    } else if (ent->isKindOf(AcDbArc::desc())) {
        if ((es = intArc(this, AcDbArc::cast(ent), intType,
            &projPlane, points)) != Acad::eOk)
        {
            return es;
        }
    } else if (ent->isKindOf(AcDbCircle::desc())) {
        if ((es = intCircle(this, AcDbCircle::cast(ent),
            intType, &projPlane, points)) != Acad::eOk)
        {
            return es;
        }
    } else if (ent->isKindOf(AcDb2dPolyline::desc())) {
        if ((es = intPline(this, AcDb2dPolyline::cast(ent),
            intType, &projPlane, points)) != Acad::eOk)
        {
            return es;
        }
    } else if (ent->isKindOf(AcDb3dPolyline::desc())) {
        if ((es = intPline(this, AcDb3dPolyline::cast(ent),
            intType, &projPlane, points)) != Acad::eOk)
        {
            return es;
        }
    } else {
        AcGePoint3dArray vertexArray;
        if ((es = getVertices3d(vertexArray))
            != Acad::eOk)
        {
            return es;
        }
        if (intType == AcDb::kExtendArg
            || intType == AcDb::kExtendBoth)
        {
            intType = AcDb::kExtendThis;
        }
```

```
AcDbLine *pAcadLine;

for (int i = 0; i < vertexArray.length() - 1; i++) {
    pAcadLine = new AcDbLine();
    pAcadLine->setStartPoint(vertexArray[i]);
    pAcadLine->setEndPoint(vertexArray[i + 1]);
    pAcadLine->setNormal(normal());
    if ((es = ent->intersectWith(pAcadLine, intType,
        projPlane, points)) != Acad::eOk)
    {
        delete pAcadLine;
        return es;
    }
    delete pAcadLine;
}

// All the points that we selected in this process are on
// the other curve; we are dealing with apparent
// intersection. If the other curve is 3D or is not
// on the same plane as poly, the points are not on
// poly.
//
// In this case, we need to do some more work. Project the
// points back onto the plane. They should lie on
// the projected poly. Find points on real poly
// corresponding to the projected points.
//
AcGePoint3d projPt, planePt;
AcGePoint3dArray pts;
AcGeLine3d line;

AcGePlane polyPlane;
AcDb::Planarity plnrty;
getPlane(polyPlane,plnrty);

for (i = 0; i < points.length(); i++) {
    // Define a line starting from the projPt and
    // along the normal.  Intersect the polygon with
    // that line. Find all the points and pick the
    // one closest to the given point.
    //
    projPt = points[i].orthoProject(projPlane);
    line.set(projPt, projPlane.normal());

    if ((es = intLine(this, line, pts))
        != Acad::eOk)
    {
        return es;
    }
```

```
            planePt = projPt.project(polyPlane,
                projPlane.normal());
            points[i] = pts[0];
            double length = (planePt - pts[0]).length();
            double length2;

            for (int j = 1; j < pts.length(); j++) {
                if ((length2 = (planePt - pts[j]).length())
                    < length)
                {
                    points[i] = pts[j];
                    length = length2;
                }
            }
        }
    }
    return es;
}
```

Intersecting a Custom Entity with Another Entity

ObjectARX is an open architecture where multiple applications can implement their own custom entities. It's possible that multiple applications will be loaded at the same time in an AutoCAD session. The user might select your custom entity in an operation that causes it to intersect with another custom entity that you are not aware of. The following guidelines should help you implement the `intersectWith()` function of your custom entity.

- Each custom entity is expected to be able to intersect with native entities. Native entities are the entities defined in AutoCAD, for example, `AcDbLine`, `AcDbEllipse`, and `AcDbSpline`.
- If the `intersectWith()` function of your custom entity is called with another entity that is not a native entity, you need to explode your custom entity (for example, by using the `explode()` function) to a set of recognizable native entities, then turn around and call `intersectWith()` on the entity that came in as an argument to your `intersectWith()` function. Because everyone is expected to be able to intersect with native entities, the entity in the argument would be able to intersect with your exploded version.

During this process, you need to be careful about how you call the `intersectWith()` function of the argument and how you interpret the points that are the results of intersection. For example, if the intersection type was `kExtendArg`, you would want to change it to `kExtendThis` before calling `intersectWith()` on the argument. Similarly, if the intersection is an apparent intersection on a plane of projection, the points returned from the

`intersectWith()` call on the argument entity will be on the argument entity, not necessarily on your entity. You're supposed to return the intersection points on your entity; therefore, you need to project the points back onto the projection plane (where they will lie on your projected entity) and then project them back onto your entity before returning.

Exploding an Entity

You must override the `explode()` function of a custom entity for the AutoCAD commands BHATCH and EXPLODE to work. Your `explode()` function should break the entity down into less complex entities. If the resulting entities are not native entities, your function should return `eExplodeAgain`. This will cause BHATCH to recursively call the `explode()` function on the entities that you return, until they have been reduced to native entities. The native entities upon which BHATCH can operate directly are `AcDb2dPolyline`, `AcDb3dPolyline`, `AcDbPolyline`, `AcDbText`, `AcDbMText`, `AcDbShape`, `AcDbTrace`, `AcDbSolid`, `AcDbFace`, `AcDbViewport`, `AcDbFcf`, `AcDbDimension`, `AcDbRegion`, `AcDbBlockReference`, and `AcDbHatch`.

Extending Entity Functionality

ObjectARX protocol extension is a flexible mechanism that can be used to add functionality to existing ObjectARX classes at runtime. Protocol extension allows you to add functions to an existing class without redefining the class and recompiling the application. (See chapter 19, "Protocol Extension.")

The `AcDbEntity` class is associated with a default protocol extension class to provide base class support for the MATCHPROP command. The default protocol extension class allows you to copy color, layer, linetype, and linetype scale properties from one entity to another. It is recommended that `AcDbMatchProperties` be implemented as a protocol extension class for all custom objects derived from `AcDbEntity`, to provide full support for MATCHPROP. If the default protocol extension class is overridden with `AcDbMatchProperties`, it must include functions to copy the base class properties as well.

Using AcEdJig

The `AcEdJig` class is used to perform drag sequences, usually to acquire, create, edit, and add a new entity to the database. If you are deriving a new entity class, you will usually want to implement your own version of `AcEdJig`. This class enables the AutoCAD user to define certain aspects of an entity using a pointing device, and it gives the programmer access to the AutoCAD drag mechanism. (The class takes its name from "jig," a device used to hold a machine part that is being bent or molded in place.)

Each time the user moves the pointing device, your application acquires a geometric value and you need to provide graphical feedback for the pointing device event. This feedback consists of two elements:

- A cursor of the specified type
- Entity graphics, returned by your `AcEdJig` object

`AcEdJig` is generally used on entities that do not reside in the database. It operates on a single entity. Do not use `AcEdJig` to operate on complex entities such as polylines.

Deriving a New Class from AcEdJig

To implement a drag sequence for your new entity, you must derive a new class from `AcEdJig` and override the following member functions:

- `AcEdJig::sampler()`, which acquires a geometric value (an angle, a distance, or a point)
- `AcEdJig::update()`, which analyzes the geometric value and stores it or updates the entity
- `AcEdJig::entity()`, which returns a pointer to the entity to be regenerated

General Steps for Using AcEdJig

`AcEdJig` is designed to control a drag sequence, by supplying graphical feedback specified by a cursor type and a single entity.

To use the AcEdJig class

1 Create an instance of your derived class of `AcEdJig`.
2 Establish your prompt text with the `AcEdJig::setDispPrompt()` function.
3 Call the `AcEdJig::drag()` function, which controls the drag loop and in turn calls the `sampler()`, `update()`, and `entity()` functions until the user ends the drag sequence.
4 Check within the `sampler()` function:

   If you are using a prompt with keywords, invoke the `AcEdJig::setKeywordList()` function.

   If you want to set a special cursor type, call the `AcEdJig::setSpecialCursorType()` function. (This step is optional and can typically be omitted.)

   If desired, place limitations on the drag sequence and the return value using the `AcEdJig::setUserInputControls()` function.
5 Check the return status from the `AcEdJig::drag()` function and commit the changes of the drag sequence. If the user canceled or aborted the process, perform the appropriate cleanup.

Setting Up Parameters for the Drag Sequence

Before you call the `AcEdJig::drag()` function, you need to set the display prompt for the drag sequence.

The display prompt is the text shown on the command line during the drag sequence. Use the following function to set the display prompt:

```
void
AcEdJig::setDispPrompt(const char* prompt);
```

Drag Loop

After you have set the display prompt for the drag sequence, you call the `AcEdJig::drag()` function, which performs the drag loop until the user presses ENTER or the space bar, or picks with the pointing device. The following list describes the sequence of the drag loop:

1 The drag loop receives an event.
2 It calls the `AcEdJig::sampler()` function. The `sampler()` function sets up the keyword list (if any) with a call to the `AcEdJig::setKeywordList()` function, a special cursor type (if desired) with a call to the `AcEdJig::setSpecialCursorType()` function, and any user input controls with a call to the `AcEdJig::setUserInputControls()` function. Next, it calls one of the `acquireXXX()` functions to obtain a geometric value (an angle,

distance, or point). The function always returns immediately after polling the current pointing device position.

3. Your `sampler()` function should check to see if there is any change in the geometric value sampled. If there is no change, your `sampler()` function should return `kNoChange` and return to step 1. This will allow the image to complete its last update on screen. This is especially important for images containing curves.
4. Even if the geometric value sampled has changed, your `sampler()` function can return `kNoChange` (so that the image is not updated) and return to step 1. If the sampled value has changed and the image needs to be updated, proceed to step 5.
5. The dragger calls the `AcEdJig::update()` function, using the acquired geometric value to update the entity.
6. The dragger then calls the `AcEdJig::entity()` function, passing in a pointer to be set to the address of the entity to be regenerated. Next, the dragger calls the `worldDraw()` function on the entity to regenerate it.
7. Return to step 1 unless the current dragger event was generated by selecting with the pointing device, pressing CANCEL, or issuing a string termination character to end dragging.

The following flowchart shows these steps.

Implementing the sampler(), update(), and entity() Functions

Within the `sampler()` function, you need to set up certain parameters for the drag sequence: the keyword list, the display prompt, the cursor type, and user input controls. The following sections describe each of these parameters in more detail.

Keyword List

If you have keywords that are meaningful in the drag sequence, use the following function to specify them:

```
void
AcEdJig::setKeywordList(const char* kyWdList);
```

The keyword list is a single string in which each keyword is separated from the others by spaces. The required characters are capitalized, and the remainder of each keyword is lowercase.

For example, "`Close Undo`" specifies two keywords. The `DragStatus` enum associates values with each keyword. The first keyword is `kKW1`, the second is `kKW2`, and so on. When you implement your `AcEdJig` class, you can use these return values in your implementations of the `sampler()`, `update()`, and `entity()` functions.

Display Prompt

The display prompt is the text shown on the command line during the drag sequence. Use the following function to set the display prompt:

```
void
AcEdJig::setDispPrompt(const char* prompt);
```

Cursor Types

If you want to set a special cursor type, use the following function:

```
void
AcEdJig::setSpecialCursorType(AcEdJig::CursorType);
```

The `CursorType` can be one of the values in the following table:

Cursor types	
Cursor	**Description**
kCrosshair	Crosshairs aligned with the user coordinate system (UCS)
kRectCursor	Rectangular window cursor aligned with the display coordinate system
kRubberBand	Same as kCrosshair, except also displays a rubber band from the base point
kTargetBox	OSNAP cursor; similar to kEntitySelect cursor, except its size is controlled by the system variable $APERTURE
kCrosshairNoRotate	Crosshairs aligned with the display coordinate system
kInvisible	No cursor graphics; only entity graphics are displayed
kEntitySelect	Single entity pick box; the entity is not actually selected in this case. Entity selection is handled with acedSSGet()
kParallelogram	Rectangle aligned with the UCS (can be a parallelogram on the display)
kEntitySelectNoPersp	Same as kEntitySelect, except the pick box is suppressed in perspective view; used when a precise geometric point is needed along with the picked entity
kPkfirstOrGrips	Default cursor; what the cursor looks like "between" commands
kArrow	Displays the arrow cursor used for dialog boxes in AutoCAD

This step is optional. The `acquirePoint()` functions allow you to specify this alternate cursor. Setting the cursor type for the `acquireDist()` and `acquireAngle()` functions has no effect. The `acquireXXX()` functions will select a cursor for you if you don't explicitly specify one.

User Input Controls

Use the following function to change the default user input controls:

```
void
setUserInputControls(AcEdJig::UserInputControls uic);
```

The user input controls put limitations on the drag sequence or the type of acceptable return value (for example, by not allowing negative responses, by not allowing a zero response, or by restricting the input value to a 2D coordinate). They also specify how various user actions affect the drag sequence. For example, `kAcceptMouseUpAsPoint` specifies that releasing the mouse button indicates the input value.

The user input controls can be one of the following values:

- `kGovernedByOrthoMode`
- `kNullResponseAccepted`
- `kDontEchoCancelForCtrlC`
- `kDontUpdateLastPoint`
- `kNoDwgLimitsChecking`
- `kNoZeroResponseAccepted`
- `kNoNegativeResponseAccepted`
- `kAccept3dCoordinates`
- `kAcceptMouseUpAsPoint`
- `kAnyBlankTerminatesInput`
- `kInitialBlankTerminatesInput`

Once you have established the keyword list, cursor type, and user input controls, your `sampler()` function should call one of the following functions of `AcEdJig` to obtain an angle, a distance, or a point:

```
DragStatus
AcEdJig::acquireAngle(double &ang);
```

```
DragStatus
AcEdJig::acquireAngle(
    double &ang,
    const AcGePoint3d &basePt);
```

```
DragStatus
AcEdJig::acquireDist(double &dist);
```

```
DragStatus
AcEdJig::acquireDist(
    double &dist,
    const AcGePoint3d &basePt);
```

```
DragStatus
AcEdJig::acquirePoint(AcGePoint3d &point);
```

```
DragStatus
AcEdJig::acquirePoint(
    AcGePoint3d &point,
    const AcGePoint3d &basePt);
```

After invoking the `sampler()` function, you can perform any further analysis on the obtained geometric value and drag status. You will also want to cache the return value in a static variable for access in your `update()` or `entity()` functions.

The `update()` function is typically where you modify the entity, usually by applying a transformation to a source entity.

The `entity()` function returns a pointer to the entity to be regenerated.

Adding the Entity to the Database

When the entity has been fully updated and the drag sequence has ended, use the `AcEdJig::append()` function to add the entity to the current space block table record. (You can also use the standard functions for appending the object to a block table record.)

Sample Code

This example creates a class that enables the user to create an ellipse by picking its center point and then dragging to select the desired major axis and minor axis lengths. During the drag operations, the user will be able to see what the ellipse looks like at any time.

NOTE If the user tries to make the minor axis longer than the major axis, the ellipse will end up as a circle because the radius ratio cannot be larger than 1.0.

```
class AsdkEllipseJig : public AcEdJig
// This class allows the user to create an ellipse by
// picking its center point and then dragging to select the
// desired major axis and minor axis lengths.  During the
// drag operations, the user will be able to visually see
// what the ellipse looks like at any time.
//
{
public:
    AsdkEllipseJig(const AcGePoint3d&, const AcGeVector3d&);
    void doIt();
    virtual DragStatus sampler();
    virtual Adesk::Boolean update();
    virtual AcDbEntity* entity() const;
private:
    AcDbEllipse *mpEllipse;
    AcGePoint3d mCenterPt, mAxisPt;
    AcGeVector3d mMajorAxis, mNormal;
    double mRadiusRatio;
    int mPromptCounter;
};
```

```
// The following defines the constructor that accepts a point to be
// used as the centerpoint of the ellipse and the current UCS normal
// vector to be used as the normal for the ellipse.  It also
// initializes the radius ratio to a small value so that during
// selection of the major axis, the ellipse will appear as a line.
// The prompt counter is also initialized to 0.
//
AsdkEllipseJig::AsdkEllipseJig(
    const AcGePoint3d& pt,
    const AcGeVector3d& normal)
: mCenterPt(pt),
  mNormal(normal),
  mRadiusRatio(0.00001),
  mPromptCounter(0)
{ }

// This function creates an AcDbEllipse object and gets the
// jig started acquiring the necessary info to properly fill
// it in.
//
void
AsdkEllipseJig::doIt()
{
    mpEllipse = new AcDbEllipse;

    // Get the major axis vector from the user.
    // At this time, mPromptCounter == 0.
    //
    setDispPrompt("\nEllipse major axis: ");
    AcEdJig::DragStatus stat = drag();

    // Get the ellipse's radius ratio.
    //
    mPromptCounter++;   // now == 1
    setDispPrompt("\nEllipse minor axis: ");
    stat = drag();

    // Now add the ellipse to the database's current space.
    //
    append();
}

// This function is called by the drag function to
// acquire a sample input.
//
AcEdJig::DragStatus
AsdkEllipseJig::sampler()
{
    DragStatus stat;
    setUserInputControls((UserInputControls)
        (AcEdJig::kAccept3dCoordinates
         | AcEdJig::kNoNegativeResponseAccepted
         | AcEdJig::kNoZeroResponseAccepted));
    if (mPromptCounter == 0) {
        // Aquire the major axis endpoint.
        //
```

```
        // If the newly acquired point is the same as it was
        // in the last sample, then we return kNoChange so the
        // AsdkEllipseJig::update() function will not be called
        // and the last update call will be able to finish, thus
        // allowing the ellipse to fully elaborate.
        //
        static AcGePoint3d axisPointTemp;
        stat = acquirePoint(mAxisPt, mCenterPt);
        if (axisPointTemp != mAxisPt)
            axisPointTemp = mAxisPt;
        else if (stat == AcEdJig::kNormal)
            return AcEdJig::kNoChange;
    }
    else if (mPromptCounter == 1) {
        // Aquire the distance from ellipse center to minor
        // axis endpoint. This will be used to calculate the
        // radius ratio.
        //
        // If the newly acquired distance is the same as it was
        // in the last sample, then we return kNoChange so the
        // AsdkEllipseJig::update() function will not be called
        // and the last update call will be able to finish, thus
        // allowing the ellipse to fully elaborate.
        //
        static double radiusRatioTemp = -1;
        stat = acquireDist(mRadiusRatio, mCenterPt);
        if (radiusRatioTemp != mRadiusRatio)
            radiusRatioTemp = mRadiusRatio;
        else if (stat == AcEdJig::kNormal)
            return AcEdJig::kNoChange;
    }
    return stat;
}

// This function is called to update the entity based on the
// input values.
//
Adesk::Boolean
AsdkEllipseJig::update()
{
    switch (mPromptCounter) {
    case 0:
        // At this time, mAxis contains the value of one
        // endpoint of the desired major axis.  The
        // AcDbEllipse class stores the major axis as the
        // vector from the center point to where the axis
        // intersects the ellipse path (such as half of the true
        // major axis), so we already have what we need.
        //
        mMajorAxis = mAxisPt - mCenterPt;
        break;
```

```
    case 1:
        // Calculate the radius ratio.  mRadiusRatio
        // currently contains the distance from the ellipse
        // center to the current pointer position.  This is
        // half of the actual minor axis length.  Since
        // AcDbEllipse stores the major axis vector as the
        // vector from the center point to the ellipse curve
        // (half the major axis), to get the radius ratio we
        // simply divide the value currently in mRadiusRatio
        // by the length of the stored major axis vector.
        //
        mRadiusRatio = mRadiusRatio / mMajorAxis.length();
        break;
    }

    // Now update the ellipse with the latest setting.
    //
    mpEllipse->set(mCenterPt, mNormal, mMajorAxis,
        mRadiusRatio);
    return Adesk::kTrue;
}

// This function must be implemented to return a pointer to
// the entity being manipulated by the jig.
//
AcDbEntity*
AsdkEllipseJig::entity() const
{
    return mpEllipse;
}

// This function uses the AcEdJig mechanism to create and
// drag an ellipse entity.  The creation criteria are
// slightly different from the AutoCAD command.  In this
// case, the user selects an ellipse center point and
// drags to visually select the major and minor axes
// lengths.  This sample is somewhat limited; if the
// minor axis ends up longer than the major axis, then the
// ellipse will just be round because the radius ratio
// cannot be greater than 1.0.
//
void
createEllipse()
{
    // First, have the user select the ellipse center point.
    // We don't use the jig for this because there is
    // nothing to see yet.
    //
    AcGePoint3d tempPt;
    struct resbuf rbFrom, rbTo;
    acedGetPoint(NULL, "\nEllipse center point: ",
        asDblArray(tempPt));

    // The point we just got is in UCS coordinates, but
    // AcDbEllipse works in WCS, so convert the point.
    //
```

```
    rbFrom.restype = RTSHORT;
    rbFrom.resval.rint = 1; // from UCS
    rbTo.restype = RTSHORT;
    rbTo.resval.rint = 0; // to WCS
    acedTrans(asDblArray(tempPt), &rbFrom, &rbTo,
        Adesk::kFalse, asDblArray(tempPt));

    // Now you need to get the current UCS z-Axis to be used
    // as the normal vector for the ellipse.
    //
    AcGeVector3d x =
      acdbHostApplicationServices()->workingDatabase()->ucsxdir();
    AcGeVector3d y =
      acdbHostApplicationServices()->workingDatabase()->ucsydir();
    AcGeVector3d normalVec = x.crossProduct(y);
    normalVec.normalize();

    // Create an AsdkEllipseJig object passing in the
    // center point just selected by the user and the normal
    // vector just calculated.
    //
    AsdkEllipseJig *pJig
        = new AsdkEllipseJig(tempPt, normalVec);

    // Now start up the jig to interactively get the major
    // and minor axes lengths.
    //
    pJig->doIt();

    // Now delete the jig object, since it is no longer needed.
    //
    delete pJig;
}

void
initApp()
{
    acedRegCmds->addCommand("ASDK_VISUAL_ELLIPSE",
        "ASDK_VELLIPSE", "VELLIPSE", ACRX_CMD_MODAL,
        createEllipse);
}

void
unloadApp()
{
    acedRegCmds->removeGroup("ASDK_VISUAL_ELLIPSE");
}
```

```
extern "C" AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
    }
    return AcRx::kRetOK;
}
```

Part IV

Specialized Topics

Proxy Objects

In This Chapter

14

- Proxy Objects Defined
- Proxy Object Life Cycle
- User Encounters with Proxy Objects
- Displaying Proxy Entities
- Editing Proxy Entities
- Unloading an Application

This chapter describes proxy objects and the conditions of their creation. It also discusses user encounters with proxies, displaying proxy entities, and editing proxy entities. The effect of unloading an application on custom objects and entities is discussed as well.

Proxy Objects Defined

A proxy object is an object AutoCAD creates in memory as a surrogate data holder for a custom ObjectARX object. AutoCAD automatically creates proxy objects when the application that defines the object is not loaded.

Proxies are created for both objects and entities. AutoCAD uses proxy objects to provide read access to data in a custom object derived from `AcDbObject` or `AcDbEntity`. Proxy objects also provide controlled edit capabilities for that data. The parent application determines the extent of those edit capabilities with the `PROXY_FLAGS` argument of the `ACRX_DXF_DEFINE_MEMBERS` macro.

The proxy object class `AcDbProxyObject` is derived from `AcDbObject`, and the proxy entity class `AcDbProxyEntity` is derived from `AcDbEntity`. Both are abstract base classes that cannot be instantiated and are included in the ObjectARX API.

Proxy objects convert back to the original custom object whenever the parent application is loaded. For example, if proxy objects are created at the beginning of a drawing session, and the parent application is subsequently loaded, the proxies are restored to custom objects.

Under special circumstances, proxies are written to files, but proxy objects usually exist only in memory.

ObjectARX developers can affect the creation and control the modification of proxies by using the `ACRX_DXF_DEFINE_MEMBERS` macro and the demand loading features in AutoCAD (see "Demand Loading" on page 45). In addition, they can use functions of the proxy object classes in their own applications to manage proxies that AutoCAD creates for the custom objects of other applications.

Proxy Object Life Cycle

Proxy objects are created by AutoCAD when it reads a file containing custom objects that cannot be instantiated. Custom objects cannot be instantiated when the parent application is not loaded and cannot be demand loaded. If the parent application is subsequently loaded during the drawing session (by command, for example), AutoCAD automatically converts the proxy objects to custom objects.

A proxy may be thought of as a wrapper containing a custom object. A proxy provides AutoCAD with access to the base class elements (such as color and layer) of the custom object, and it encapsulates the custom class's data mem-

bers that cannot be accessed in the absence of the parent application. Under most circumstances, the proxy wrapper is shed when the drawing database is written to a file. The same binary object that was read in is written out. If the save operation involves converting the file type between DWG and DXF (for which the conversion function of the parent application is not present), the proxy wrapper is saved with the custom binary data as well.

When the parent application is not loaded before writing to a file, the data is handled as follows:

- If the input and output files are the same file type (both DWG or DXF), no translation operation is necessary, and the same data that was read in is written out. The data stored in the proxy object is written to the output file.
- If the input and output files differ in file type (that is, DWG in and DXF out, or vice versa), the format cannot be translated because the translation function defined by the parent application is not present. The entire proxy object is therefore written to the output file. When the file is subsequently read by AutoCAD, the proxy will either convert to a custom object (in the presence of the parent application), or remain a proxy in memory (in the absence of the parent application).

User Encounters with Proxy Objects

An AutoCAD message notifies users of the creation of proxy objects. Users can read about proxy entities using the LIST command. They can also encounter proxies because of the manner in which the entities are displayed and the way they respond to editing.

AutoCAD displays a message immediately after any command that causes a proxy object to be created (for example, DXFIN, XREF, INSERT). The message includes the number of proxy entities created with visual representation, the number of proxy entities created that cannot be displayed because of the absence of the parent application, and the number of proxy objects created without visual representation.

A LIST operation performed on a proxy object identifies the application that created the custom object, and identifies the object as being either of type `ACAD_PROXY_OBJECT` or `ACAD_PROXY_ENTITY`.

Users can control the display of proxy objects with the PROXYSHOW system variable, which has the following options:

1	None shown
2	Graphical representation
3	Bounding box

Displaying Proxy Entities

AutoCAD cannot display a proxy entity using the object's `worldDraw()` or `viewportDraw()` functions because the parent application is absent. It uses information in the entity's graphics metafile, which contains data derived from the entity's `worldDraw()` or `saveAs()` function when the drawing was last saved (with the SAVE or SAVEAS command). The display format is either that of the entity itself, or a bounding box indicating the volume inhabited by the entity. The format is determined by the setting of the PROXYGRAPHICS system variable at the time the drawing is saved. The bounding box format is useful primarily to minimize the size of the output file when the data for custom entities is large.

Editing Proxy Entities

The extent to which proxy entities can be edited is determined by the parent application. This determination is made when the class is created with the `ACRX_DXF_DEFINE_MEMBERS` macro. The `PROXY_FLAGS` argument defines the types of edits that can be made to the entity if it becomes a proxy. The valid options for `PROXY_FLAGS`, and their associated values, are listed in the following table.

Proxy flags options

Option	Value
kNoOperation	0
kEraseAllowed	0x1
kTransformAllowed	0x2
kColorChangeAllowed	0x4

Proxy flags options (*continued*)	
Option	**Value**
kLayerChangeAllowed	0x8
kLinetypeChangeAllowed	0x10
kLinetypeScaleChangeAllowed	0x20
kVisibilityChangeAllowed	0x40
kAllAllowedBits	0x7F

Note that `kNoOperation` means none of the other options listed here.

You can logically OR `PROXY_FLAG` options to permit a combination of editing operations.

As proxy entities only encapsulate data below the `AcDbEntity` base class level, any changes made to color, layer, linetype, linetype scale, and visibility will be written out as part of the proxy entity data. Rigid body transformations (such as move, scale, and rotate) cannot be applied until the parent application is present. When a transformation is applied to a proxy, the transformation is made to the graphics metafile, and a copy of the transformation matrix is saved in a custom record in the proxy entity's extension dictionary. If multiple transformations are performed, the matrix is updated to reflect the cumulative transformation. When the custom entity is returned to memory with its parent application, AutoCAD calls the entity's `transformBy()` function, passes it the transformation matrix data, and removes the custom data storage record from the extension dictionary. In effect, the transformation is deferred until the parent application is present to apply the transformation to the custom entity.

Unloading an Application

When an application is unloaded and the appropriate cleanup operations have been performed, custom objects and entities are transformed into proxy objects. For this to occur, all custom classes must be removed from ObjectARX at unload time with the `deleteAcRxClass()` function. For a description of the requirements for making an application "unloadable," see chapter 3, "ObjectARX Application Basics."

Notification

This chapter describes how you can create reactors that respond to different event types and register the reactors with the appropriate objects to receive notification.

In This Chapter

15

- Notification Overview
- Using Reactors
- Notification Use Guidelines

Notification Overview

When an event occurs in the system, certain objects, called notifiers, automatically relay the event to other objects. For example, when a user copies, erases, or modifies an object or when a user issues an UNDO or REDO command, a corresponding notification for each event is automatically triggered.

The objects receiving the events are called reactors. A reactor must be explicitly added to a notifier's reactor list before it can receive events from the notifier. A given notifier can have a number of reactors in its reactor list. The reactor's class definition includes various notification functions. When an event occurs, the notifier automatically invokes the corresponding notification function of each reactor in its reactor list.

To use a reactor in an application

1. Derive a new reactor class and implement the notification functions for the events your reactor will respond to.
2. Instantiate the reactor.
3. Add the reactor to the reactor list of the notifier.

When finished using the reactor

1. Remove the reactor from the reactor lists of all notifiers to which it has been added.
2. Delete the reactor (unless it is a database-resident object).

Using reactors requires creating subclasses of reactor classes or of `AcDbObject` classes. This chapter assumes you are familiar with the material presented in chapter 11, "Deriving a Custom ObjectARX Class," and chapter 12, "Deriving from AcDbObject."

Reactor Classes

Reactor classes are derived from `AcRxObject`, not `AcDbObject`. Because these reactors are not database objects, ownership does not apply to them and they don't have object IDs.

Different kinds of reactors receive different types of notification events. A database reactor (derived from `AcDbDatabaseReactor`) receives events related to the state of the database—for example, when an object is appended to the database, modified in the database, or erased. The reactor's notifier is the database, so it is added to the reactor list of the `AcDbDatabase`. An object reactor (derived from `AcDbObjectReactor`) responds to events at the object level,

such as copying, erasing, or modifying an object. It can be added to the reactor list of any `AcDbObject`. An editor reactor (derived from `AcEditorReactor`) responds to AutoCAD-specific events such as loading and unloading a drawing, starting or ending a command, and other kinds of user interaction. The `AcEditor` object is the only notifier for an `AcEditorReactor`. The following is the class hierarchy for reactor classes:

```
AcRxObject
    ├── AcApDocManagerReactor
    ├── AcApLongTransactionReactor
    ├── AcDbDatabaseReactor
    ├── AcDbObjectReactor
    │    └── AcDbEntityReactor
    │         ├── AcDbRasterImageDefFileAccessReactor
    │         └── AcDbRasterImageDefTransReactor
    ├── AcEdInputContextReactor
    ├── AcRxDLinkerReactor
    ├── AcRxEventReactor
    │    └── AcEditorReactor
    └── AcTransactionReactor
```

Types of Object Reactors

The reactor classes shown above are also referred to as transient reactor classes. If you want your program to receive event notification, you'll usually use transient reactors, which monitor events that happen to database objects. They can also monitor database events, user interaction, and other system events while an application is running.

Another kind of reactor, called a persistent reactor, uses a database object (an instance of class `AcDbObject` or a derived class) as a reactor. Database objects can receive as well as send notification. Persistent reactor dependencies within the database are part of the database, so they are preserved in DWG and DXF files and are reestablished when a drawing is loaded.

To use an AcDbObject as a reactor

1. Derive a new `AcDbObject` class and implement the notification functions for the events your object will respond to.
2. Instantiate the object reactor.
3. Add the object reactor to the database and give it an owner, preferably a container object, so that it is filed out correctly.
4. Add the object reactor to the notifier's reactor list using the `addPersistentReactor()` function. This function requires you to pass in the object ID of the object reactor you created in step 2.

AutoCAD will delete the object reactor, because it is a database object.

NOTE When you copy an object, any persistent reactors attached to the object are copied as well. Transient reactor attachments are not copied when an object is copied.

Using Reactors

To use a transient reactor, derive a new class from one of the following base classes:

AcRxDLinkerReactor

Monitors ObjectARX application loading and unloading.

AcEditorReactor

Monitors AutoCAD-specific events such as commands and AutoLISP evaluations.

AcDbDatabaseReactor

Monitors creation, modification, and erasure of database objects.

AcTransactionReactor

Monitors events related to the transaction manager—start, abort, or end of a transaction.

AcDbObjectReactor

Monitors events pertaining to a specific database object—creation, modification, erasure.

AcDbEntityReactor

Monitors an extra, entity-specific event, such as modified graphics.

In most cases, only standard C++ techniques are needed for creating new transient reactor classes. The ObjectARX macros, which create a class descriptor object for the new reactor class, are usually not used to derive from these reactor classes.

Each parent class contains a set of virtual notification functions that can be implemented by your new derived class. For example, the `AcDbObjectReactor` class contains the following notification functions that respond to object-related events:

- `cancelled()`
- `copied()`
- `erased()`
- `goodbye()`
- `openedForModify()`
- `modified()`
- `subObjModified()`
- `modifyUndone()`
- `modifiedXData()`
- `unappended()`
- `reappended()`
- `objectClosed()`

Each of these functions requires a pointer to the notifier of the event. The base class, `AcDbObjectReactor`, has `NULL` implementations for all of these functions. In your derived reactor class, implement the functions corresponding to the type of notifications you are interested in. Then instantiate the reactor and add it to any number of database objects using the `AcDbObject::addReactor()` function. To add or delete a transient reactor to a notifier object, the object can be open in any state (read, write, or notify). Adding or deleting a transient reactor is not monitored by the undo mechanism. (For persistent reactors, the notifier object must be opened for write, and adding or removing the reactors *is* monitored by the undo mechanism.) Because you created the transient reactor object, you are also responsible for deleting it.

When an object is erased, for example, it calls the corresponding `erased()` notification function on each reactor in its list. If you have implemented an `erased()` function for your reactor, that function will be called by the database object, and you can then take whatever special action is appropriate for your application when an object is erased.

AcDbObject and Database Notification Events

When you receive `erased()` notification on a database object, the object is marked as erased but is still part of the database. When you receive `unappended()` notification, the object has been marked unappended and is not part of the database unless it is reappended. The `goodbye()` notification on an object is sent just before it goes away completely. This notification signals that the object is about to be removed from the database and deleted from memory.

You may want to remove your reactor from an object when you receive `erased()` or `unappended()` notification. However, if you remove the reactor at this point, you won't receive `reappended()` or `unerased()` notification for that object. To monitor these events, use the equivalent notifications on the database, not just the object:

```
AcDbDatabaseReactor::objectErased()
AcDbDatabaseReactor::objectUnappended()
AcDbDatabaseReactor::objectReappended()
```

Custom Notifications

When modifications are committed on an object, the object is closed, which invokes the `subClose()` virtual function of `AcDbObject`. In the override of this function in your custom class, you can notify others that you are closing after modification. These notifications should be your custom notification in the form of custom functions on your class. Do not use the notifications provided on `AcDbObjectReactor` for this purpose.

Using an Editor Reactor

The `AcEditorReactor` class provides many functions for responding to various events. A few of these functions are `beginClose()`, `beginDxfIn()`, `dxfInComplete()`, `beginSave()`, and `saveComplete()`. No AutoLISP interaction can be performed within the notification function.

See chapter 18, "Deep Cloning," for a discussion of editor reactor functions relating to the deep clone and wblock clone operations.

Using a Database Reactor

The following example uses a reactor derived from `AcDbDatabaseReactor` to keep track of the number of objects currently in the database. It implements three notification functions for the reactor class: `objectAppended()`, `objectModified()`, and `objectErased()`. The `watch_db()` function adds the reactor to the current database. The `clear_reactors()` function removes the reactor from the database and deletes the database reactor.

```
class AsdkDbReactor;
long gEntAcc = 0;            // Global entity count
AsdkDbReactor *gpDbr = NULL; // Pointer to database reactor

// Custom AcDbDatabaseReactor class for database
// event notification.
//
class AsdkDbReactor : public AcDbDatabaseReactor
{
public:
    virtual void objectAppended(const AcDbDatabase* dwg,
        const AcDbObject* dbObj);
    virtual void objectModified(const AcDbDatabase* dwg,
        const AcDbObject* dbObj);
    virtual void objectErased(const AcDbDatabase* dwg,
        const AcDbObject* dbObj, Adesk::Boolean pErased);
};

// Called whenever an object is added to the database.
//
void
AsdkDbReactor::objectAppended(const AcDbDatabase* db,
    const AcDbObject* pObj)
{
    printDbEvent(pObj, "objectAppended");
    acutPrintf(" Db==%lx\n", (long) db);
    gEntAcc++;
    acutPrintf("Entity Count = %d\n", gEntAcc);
}

// Called whenever an object in the database is modified.
//
void
AsdkDbReactor::objectModified(const AcDbDatabase* db,
    const AcDbObject* pObj)
{
    printDbEvent(pObj, "objectModified");
    acutPrintf(" Db==%lx\n", (long) db);
}

// Called whenever an object is erased from the database.
//
```

```
void
AsdkDbReactor::objectErased(const AcDbDatabase* db,
    const AcDbObject* pObj, Adesk::Boolean pErased)
{
    if (pErased) {
        printDbEvent(pObj, "objectErased");
        gEntAcc--;
    } else {
        printDbEvent(pObj, "object(Un)erased");
        gEntAcc++;
    }
    acutPrintf(" Db==%lx\n", (long) db);
    acutPrintf("Entity Count = %d\n", gEntAcc);
}

// Prints the message passed in by pEvent; then
// calls printObj() to print the information about
// the object that triggered the notification.
//
void
printDbEvent(const AcDbObject* pObj, const char* pEvent)
{
    acutPrintf("  Event: AcDbDatabaseReactor::%s ", pEvent);
    printObj(pObj);
}

// Prints out the basic information about the object pointed
// to by pObj.
//
void
printObj(const AcDbObject* pObj)
{
    if (pObj == NULL) {
        acutPrintf("(NULL)");
        return;
    }
    AcDbHandle objHand;
    char  handbuf[17];

    // Get the handle as a string.
    //
    pObj->getAcDbHandle(objHand);
    objHand.getIntoAsciiBuffer(handbuf);
    acutPrintf(
        "\n   (class==%s, handle==%s, id==%lx, db==%lx)",
        pObj->isA()->name(), handbuf,
        pObj->objectId().asOldId(), pObj->database());
}

// Adds a reactor to the database to monitor changes.
// This can be called multiple times without any ill
// effect because subsequent calls will be ignored.
//
```

```
void
watchDb()
{
    if (gpDbr == NULL) {
        gpDbr = new AsdkDbReactor();
    }
    acdbHostApplicationServices()->workingDatabase()->addReactor(
        gpDbr);
    acutPrintf(
        "  Added Database Reactor to "
        "acdbHostApplicationServices()->workingDatabase().\n");
}

// Removes the database reactor.
//
void
clearReactors()
{
    if (acdbHostApplicationServices()->workingDatabase() != NULL) {
        acdbHostApplicationServices()->workingDatabase(
            )->removeReactor(gpDbr);
        delete gpDbr;
        gpDbr = NULL;
    }
}

// ObjectARX entry point function
//
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppNotMDIAware(appId);
        acedRegCmds->addCommand("ASDK_NOTIFY_TEST",
            "ASDK_WATCH",
            "WATCH",
            ACRX_CMD_TRANSPARENT,
            watchDb);
        acedRegCmds->addCommand("ASDK_NOTIFY_TEST",
            "ASDK_CLEAR",
            "CLEAR",
            ACRX_CMD_TRANSPARENT,
            clearReactors);
        break;
    case AcRx::kUnloadAppMsg:
        clearReactors();
        acedRegCmds->removeGroup("ASDK_NOTIFY_TEST");
        break;
    }
    return AcRx::kRetOK;
}
```

Using an Object Reactor

To cause one database object to react to another database object

1. Derive a class from `AcDbObject` (or any of its subclasses).
2. Implement the notification functions.
3. Instantiate an object of the class.
4. Add the object to the database, and assign an owner.
5. Add it to the notifier object with the `AcDbObject::addPersistentReactor()` function.

This mechanism allows you to define dependencies within a database that are preserved when the database is saved and recreated whenever it is reinstantiated.

Use the ObjectARX macros when you derive the new object reactor class so that a class descriptor object will be created for it. (If you don't use the ObjectARX macros, your class will inherit the class description of its parent when it is saved, and its identity will be lost when the file is read in.)

Obtaining the ID of the Object Reactor

Every database object maintains a list of reactors on itself. Some are transient reactors, and some are persistent. Transient reactors are instances of classes derived from `AcDbObjectReactor`, whereas persistent reactors are the object IDs of database-resident objects.

The following code shows how to search through the list of reactors to find your transient or persistent reactor. It is extremely important that you verify a particular entry in the reactor list to be a persistent reactor by using the `AcDbIsPersistentReactor` function. If it is a persistent reactor, you can use the appropriate function to obtain its object ID. If it is not a persistent reactor, you can cast the entry to `AcDbObjectReactor`.

```
AcDbVoidPtrArray *pReactors;
void             *pSomething;

AcDbObjectReactor *pObjReactor;
AcDbObjectId       persObjId;
AcDbObject        *pPersReacObj;

pReactors = pEnt->reactors();

if (pReactors != NULL && pReactors->length() > 0) {
    for (int i = 0; i < pReactors->length(); i++) {
        pSomething = pReactors->at(i);
```

```
// Is it a persistent reactor?
//
if (acdbIsPersistentReactor(pSomething)) {
    persObjId = acdbPersistentReactorObjectId(
        pSomething);
    acutPrintf("\n\nPersistent reactor found.");

    // Echo the keyname to the user.
    //
    char *keyname = NULL;
    getPersReactorKey(keyname, persObjId);
    if (keyname) {
        acutPrintf("\nThis is the reactor named %s",
        keyname);
        free (keyname);
    }
    // Open it up and see if it's one of ours. If it is,
    // fire the custom notification.
    //
    if ((retStat =
        acdbOpenAcDbObject(pPersReacObj,
        persObjId, AcDb::kForNotify))
        != Acad::eOk)
    {
        acutPrintf("\nFailure for"
            " openAcDbObject: retStat==%d\n",
            retStat);
        return;
    }

    AsdkPersReactor *pTmpPers;
    if ((pTmpPers =
        AsdkPersReactor::cast((AcRxObject*)
        pPersReacObj)) != NULL)
    {
        pTmpPers->custom();
    }
    pPersReacObj->close();
} else {
    // Or is it transient?
    //
    pObjReactor = (AcDbObjectReactor *)
                  (pReactors->at(i));
    acutPrintf("\n\nTransient Reactor found");

    // Report what kind we found.
    //
```

```
            if (pObjReactor->isKindOf(
                AsdkSimpleObjReactor::desc()))
            {
                acutPrintf(" of type"
                    " AsdkSimpleObjReactor");
            } else if (pObjReactor->isKindOf(
                AcDbEntityReactor::desc()))
            {
                acutPrintf(" of type"
                    " AcDbEntityReactor");
            } else if (pObjReactor->isKindOf(
                AcDbObjectReactor::desc()))
            {
                acutPrintf(" of type"
                    " AcDbObjectReactor");
            } else {
                acutPrintf(" of unknown type.");
            }
        }
    }
} else {
    acutPrintf("\nThis entity has no reactors.\n");
}
```

Example: Building in Object Dependencies

The following example shows how you can use reactors to establish dependencies among database objects. In this example, when you change one line, the other line changes.

```
class AsdkObjectToNotify : public AcDbObject
//
// AsdkObjectToNotify - customized AcDbObject for persistent
// reactor to notify.
//
{
public:
    ACRX_DECLARE_MEMBERS(AsdkObjectToNotify);
    AsdkObjectToNotify() {};
    void eLinkage(AcDbObjectId i, double f=1.0)
        {mId=i; mFactor=f; };
    void              modified(const AcDbObject*);
    Acad::ErrorStatus dwgInFields(AcDbDwgFiler*);
    Acad::ErrorStatus dwgOutFields(AcDbDwgFiler*) const;
    Acad::ErrorStatus dxfInFields(AcDbDxfFiler*);
    Acad::ErrorStatus dxfOutFields(AcDbDxfFiler*) const;
private:
    AcDbObjectId mId;
    double mFactor;
};
ACRX_DXF_DEFINE_MEMBERS(AsdkObjectToNotify, AcDbObject,
    AcDb::kDHL_CURRENT, AcDb::kMReleaseCurrent,
    0, ASDKOBJECTTONOTIFY, persreac);
```

```
// This function is called every time the line it's
// "watching" is modified.  When it's called, it opens the
// other line of the pair and changes that line's length to
// match the new length of the line that's just been
// modified.
//
void
AsdkObjectToNotify::modified(const AcDbObject* pObj)
{
    AcDbLine *pLine = AcDbLine::cast(pObj);

    if (!pLine) {
        const char* cstr = pObj->isA()->name();
        acutPrintf("This is a %s.\n", cstr);
        acutPrintf("I only work with lines. Sorry.\n");
        return;
    }

    acutPrintf("\nReactor attached to %lx calling %lx.\n",
        pLine->objectId(), mId);

    // This open will fail during notification caused by a
    // reactor being added to the entity or when this
    // notification is in reaction to a change due to the
    // other line's reactor changing this line. This will
    // properly prevent an infinite recursive loop
    // between the two lines and their reactors.
    //
    AcDbLine *pLine2;

    if (acdbOpenObject((AcDbObject*&)pLine2, mId,
        AcDb::kForWrite) == Acad::eOk)
    {
        // Get length of line entity we're being notified
        // has just been modified.
        //
        AcGePoint3d p = pLine->startPoint();
        AcGePoint3d q = pLine->endPoint();
        AcGeVector3d v = q-p;
        double len = v.length();

        // update other entity to match:
        //
        p = pLine2->startPoint();
        q = pLine2->endPoint();
        v = q-p;
        v = len * mFactor * v.normal();
        pLine2->setEndPoint(p+v);
        pLine2->close();
    }
}

// Files an object's information in.
//
```

```
Acad::ErrorStatus
AsdkObjectToNotify::dwgInFields(AcDbDwgFiler* filer)
{
    assertWriteEnabled();
    AcDbObject::dwgInFields(filer);
    filer->readItem(&mFactor);
    filer->readItem((AcDbSoftPointerId*) &mId);
    return filer->filerStatus();
}

// Files an object's information out.
//
Acad::ErrorStatus
AsdkObjectToNotify::dwgOutFields(AcDbDwgFiler* filer) const
{
    assertReadEnabled();
    AcDbObject::dwgOutFields(filer);
    filer->writeItem(mFactor);
    filer->writeItem((AcDbSoftPointerId&)mId);
    return filer->filerStatus();
}

// Files an object's information in from DXF and AutoLISP.
//
Acad::ErrorStatus
AsdkObjectToNotify::dxfInFields(AcDbDxfFiler* filer)
{
    assertWriteEnabled();
    Acad::ErrorStatus es;
    if ((es = AcDbObject::dxfInFields(filer))
        != Acad::eOk)
    {
        return es;
    }

    // Check if we're at the right subclass data marker.
    //
    if(!filer->atSubclassData("AsdkObjectToNotify")) {
        return Acad::eBadDxfSequence;
    }

    struct resbuf rbIn;
```

```
    while (es == Acad::eOk) {
        if ((es = filer->readItem(&rbIn)) == Acad::eOk) {
            if (rbIn.restype == AcDb::kDxfReal) {
                mFactor = rbIn.resval.rreal;
            } else if (rbIn.restype
                == AcDb::kDxfSoftPointerId)
            {
                // ObjectIds are filed in as ads_names.
                //
                acdbGetObjectId(mId, rbIn.resval.rlname);
            } else {   // invalid group
                return(filer->pushBackItem());
            }
        }
    }
    return filer->filerStatus();
}

// Files an object's information out to DXF and AutoLISP.
//
Acad::ErrorStatus
AsdkObjectToNotify::dxfOutFields(AcDbDxfFiler* filer) const
{
    assertReadEnabled();
    AcDbObject::dxfOutFields(filer);
    filer->writeItem(AcDb::kDxfSubclass,
        "AsdkObjectToNotify");
    filer->writeItem(AcDb::kDxfReal, mFactor);
    filer->writeItem(AcDb::kDxfSoftPointerId, mId);
    return filer->filerStatus();
}

// Creates two lines and two AsdkObjectToNotify objects and
// ties them all together.
//
void
assocLines()
{
    AcDbDatabase *pDb =
        acdbHostApplicationServices()->workingDatabase();
    AcDbObjectId aId, bId;
    AcDbLine *pLineA = new AcDbLine;
    pLineA->setDatabaseDefaults(pDb);
    pLineA->setStartPoint(AcGePoint3d(1, 1, 0));
    pLineA->setEndPoint(AcGePoint3d(2, 1, 0));
    addToModelSpace(aId, pLineA);

    acutPrintf( "Line A is %lx from 1,1 to 2,1.\n",
        pLineA->objectId());
    AcDbLine *pLineB = new AcDbLine;
    pLineB->setDatabaseDefaults(pDb);
    pLineB->setStartPoint(AcGePoint3d(1, 2, 0));
    pLineB->setEndPoint(AcGePoint3d(2, 2, 0));
    addToModelSpace(bId, pLineB);
    acutPrintf("Line B is %lx from 1,2 to 2,2.\n",
        pLineB->objectId());
```

```
// Open the named object dictionary, and check if there is
// an entry with the key "ASDK_DICT". If not, create a
// dictionary and add it.
//
AcDbDictionary *pNamedObj;
AcDbDictionary *pNameList;
pDb->getNamedObjectsDictionary(pNamedObj,
    AcDb::kForWrite);

if (pNamedObj->getAt("ASDK_DICT",
    (AcDbObject*&)pNameList, AcDb::kForWrite)
        == Acad::eKeyNotFound)
{
    pNameList = new AcDbDictionary;
    AcDbObjectId DictId;
    pNamedObj->setAt("ASDK_DICT", pNameList, DictId);
}

pNamedObj->close();

// Create the AsdkObjectToNotify for line A.
//
AsdkObjectToNotify *pObj = new AsdkObjectToNotify();
pObj->eLinkage(bId);
AcDbObjectId objId;

if ((pNameList->getAt("object_to_notify_A", objId))
    == Acad::eKeyNotFound)
{
    pNameList->setAt("object_to_notify_A", pObj, objId);
    pObj->close();
} else {
    delete pObj;
    acutPrintf("object_to_notify_A already exists\n");
}

// Set up persistent reactor link between line A
// and AsdkObjectToNotify.
//
pLineA->addPersistentReactor(objId);
pLineA->close();

// Create the AsdkObjectToNotify for line B.
//
pObj = new AsdkObjectToNotify();
pObj->eLinkage(aId);
```

```
    if ((pNameList->getAt("object_to_notify_B", objId))
        == Acad::eKeyNotFound)
    {
        pNameList->setAt("object_to_notify_B", pObj, objId);
        pObj->close();
    } else {
        delete pObj;
        acutPrintf("object_to_notify_B already exists\n");
    }

    pNameList->close();

    // Set up persistent reactor link between line B
    // and AsdkObjectToNotify.
    //
    pLineB->addPersistentReactor(objId);
    pLineB->close();
}

// Adds an entity to model space, but does not close
// the entity.
//
void
addToModelSpace(AcDbObjectId &objId, AcDbEntity* pEntity)
{
    AcDbBlockTable *pBlockTable;
    AcDbBlockTableRecord *pSpaceRecord;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);
    pBlockTable->getAt(ACDB_MODEL_SPACE, pSpaceRecord,
        AcDb::kForWrite);
    pBlockTable->close();
    pSpaceRecord->appendAcDbEntity(objId, pEntity);
    pSpaceRecord->close();
    return;
}

// This is the initialization function called from acrxEntryPoint()
// during the kInitAppMsg case.  This function is used to add
// commands to the command stack.
//
void
initApp()
{
    acedRegCmds->addCommand("ASDK_ALINES", "ASDK_ALINES",
        "ALINES", ACRX_CMD_MODAL, assocLines);
    AsdkObjectToNotify::rxInit();
    acrxBuildClassHierarchy();
}

// This is the clean-up function called from acrxEntryPoint() during
// the kUnloadAppMsg case. This function removes this application's
// command set from the command stack.
//
```

```
void
unloadApp()
{
    acedRegCmds->removeGroup("ASDK_ALINES");

    // Remove the AsdkObjectToNotify class from the ACRX
    // runtime class hierarchy. If this is done while the
    // database is still active, it should cause all objects
    // of class AsdkObjectToNotify to be turned into proxies.
    //
    deleteAcRxClass(AsdkObjectToNotify::desc());
}

// ObjectARX entry point
//
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
    }
    return AcRx::kRetOK;
}
```

Immediate versus Commit-Time Events

For `AcDbObjectReactor`, notification events can either occur immediately or be deferred until commit time. Commit time is defined as the time an object is closed if you are operating on a per-object basis, or the end of the outermost transaction if you are using the transaction model. The following events send immediate notification:

cancelled()

Notification is sent when `AcDbObject::cancelled()` is invoked.

openedForModify()

Notification is sent the first time the modification function is invoked on an object, before the object's state is changed.

copied()

Notification is sent when the object is copied.

goodbye()

Notification is sent when the object is about to be deleted from memory.

Immediate notifications are triggered at the same time as the corresponding event. For example, when `assertWriteEnabled()` is called the first time on an object, `openedForModify()` notification is immediately sent to all reactors on that object.

The following events are sent at commit time:

- `modified()`
- `subObjModified()`
- `erased()`
- `modifyUndone()`
- `modifiedXData()`
- `unappended()`
- `reappended()`
- `graphicsModified()`

The `modified()` notification of `AcDbObjectReactor` is an example of commit-time notification. Suppose an object is opened and a modification function is called on it. The modification function calls `assertWriteEnabled()` and all reactors receive the `openedForModify()` reaction. Subsequent modification functions on the object do not result in any further notification. When the object is finally closed, a `modified()` notification is sent. However, if the opener had chosen to call `cancel()` on the object instead of `close()`, a `cancelled()` notification would have been sent instead of the `modified()` notification.

When you receive a deferred notification such as `modified()` at commit time, one of the arguments is a pointer to an object. At this time, the object is in a read-only state. You are not able to modify it until the end of the commit process.

Attempting to modify an object before the commit process is finished causes AutoCAD to abort with the error message `eWasNotOpenForWrite` or `eInProcessOfCommitting`.

You can use the following functions to check that the commit process is ended before you open the object for write:

```
AcDbObjectReactor::objectClosed(AcDbObjectId objId);

AcTransactionReactor::transactionEnded(
    int numActiveAndSuccessful);
```

The `objectClosed()` notification is sent when the object is completely closed and the pointer is no longer valid. You can open the object again using the ID that is passed in the argument and operate on it. Be careful not to create infinite notification loops at this point.

In the `transactionEnded()` notification, you can use the `numActiveTransactions()` function to query the transaction manager to see how many transactions are active. If there are no active transactions, the transaction has ended and all the objects in the transaction have been committed.

Sometimes you may need to know when the outermost transaction is ending and the commit process is beginning. Use the following notification for this purpose:

```
AcTransactionReactor::endCalledOnOutermostTransaction()
```

When the outermost transaction ends, the commit process begins and `close()` is called on each object. You might receive `objectClosed()` notification as part of this close. However, it's generally best not to act immediately. Instead, wait until the whole transaction is finished before you perform any operations on these objects.

Notification Use Guidelines

While using notifications, adhere to the following guidelines. Using notifications that violate these guidelines could result in unpredictable results for your application.

- Do not rely on the sequence of notification firing.

  You can count on `commandWillStart()` being fired before `commandEnded()`, and `beginInsert()` being fired before `endInsert()`. Relying on any other sequences might result in problems for your application if the sequence is changed when new notifications are introduced, or existing ones are rearranged.
- Do not rely on the sequence of operations (function calls) between notifications.

If you tie your application to this level of detail, your application may fail in future releases.

Instead of relying on sequences, rely on notifications to indicate the state of the system. For example, when you receive `erased(kTrue)` notification on object A, it means that object A is erased. If you receive `erased()` notification on A followed by an `erased()` notification on B, it means only that both objects A and B are erased. The system will *not* guarantee that B will always be erased after A.

- Do not use any user interaction functions in your notification callback function, such as `acedCommand()`, `acedGetPoint()`, `acedGetKword()`, or any other `acedXXX()` function.

Similar interpretations apply to notifications on database reactors, editor reactors, and transaction reactors.

The Multiple Document Interface

16

AutoCAD supports a multiple document interface (MDI) that allows you to have more than one drawing loaded at once in a single AutoCAD session. This chapter describes how to work with the MDI in your ObjectARX application.

In This Chapter

- Overview
- Terminology
- SDI System Variable
- Levels of Compatibility
- Interacting with Multiple Documents
- Document Event Notification
- Application-Specific Document Objects
- Nonreentrant Commands
- Multi-Document Commands
- Disabling Document Switching
- Application Execution Context
- Database Undo and Transaction Management Facilities
- Document-Independent Databases
- An MDI-Aware Example Application

Overview

AutoCAD supports a multiple document interface, and ObjectARX applications running within AutoCAD must operate properly in the MDI environment. Three principles must be observed for an ObjectARX application to provide MDI support:

- An application must maintain document-specific state on the stack, in a database, or in a structure that can be indexed through the corresponding document pointer.
- All documents must be locked to be modified. Basic document locking is handled automatically for AutoCAD commands, ObjectARX commands, and AutoLISP functions. Modeless dialog and toolbar code, and any commands that need to work outside the active document must manually perform document locking.
- The application must maintain the relationships between documents and databases. The AutoCAD database library (AcDb) is unaware of documents and MDI, and should remain so.

Several architectural features of ObjectARX make supporting the MDI possible. These include separate execution contexts, data instances, document locking, and the document management classes. The following sections discuss these topics in more detail.

Document Execution Contexts

A separate execution context is established for each document opened in AutoCAD, to allow the command processors and Visual LISP environments for each document to execute independently of each other. Each execution context maintains its own stack. All execution contexts work on common heap and static data. Execution contexts can only be switched when the system polls for more interactive input.

Data Instances

There is a separate instance of all data elements related to the database and current command processing state for each document. This includes the command processor, input processor, Visual LISP environment, databases, selection sets, and (most, but not all) system variables. The current command processing state is maintained in a heap. In addition to these built-in system elements, all ObjectARX applications must also maintain a document-specific state either on the stack or in structures on the heap that correspond to each active document.

Each document has its own current database, plus any number of xref databases and side databases. By default, a database is associated with one document, and it participates in the undo recording and playback for that document. However, databases may also be created independently of any document, in which case their undo state is either disabled or maintained by an application's custom undo facility.

Document Locking

All documents must be locked in order to be modified. Documents may also be locked to prevent code in other execution contexts from modifying them temporarily. Documents do not have to be locked to perform query (read) operations, and they are never prevented from performing query operations on other documents. Basic document locking is handled automatically for AutoCAD commands, ObjectARX commands, and AutoLISP functions. Modeless dialog and toolbar code, and any commands that need to work outside the active document must manually perform document locking.

For more detailed information on document locking, see "Explicit Document Locking" on page 425.

Document Management Classes

ObjectARX provides a set of classes to manage documents within an ObjectARX application. Each open drawing has an associated `AcApDocument` object. The `AcApDocManager` class manages all the `AcApDocument` objects associated with an application.

AcApDocument

The `AcApDocument` object contains information such as the filename, MFC `CDocument` object, current database, and save format of the current drawing. Additionally, the `AcApDocument` object contains functions that query the status of document locking.

AcApDocManager

The `AcApDocumentManager` object contains all the document objects in an application (there is one document object for each drawing that is open and being edited). There is only one instance, which can be obtained using the macro `acDocManager()`.

AcApDocumentIterator

The `AcApDocumentIterator` class provides the ability to iterate over the set of currently open `AcApDocument` objects.

AcApDocManagerReactor

The `AcApDocManagerReactor` class provides a reactor that applications can use to track modifications in documents and switches between documents.

For more information on the document management classes, see the *ObjectARX Reference*. An example of using these classes is given later in this chapter.

Terminology

The following section defines some commonly used terms to describe the multiple document interface.

Active Document

The document that has window focus, and receives the next user input event, unless the user is switching to (activating) another document. The active document is always the current document when processing user input, but programs can temporarily change the current document during some operations.

Application

The overall running program and associated objects that are common to all open documents, such as the MFC class `CwinApp`. There is one and only one application per invocation of an executable Windows program.

Application Context

Short for "application execution context." See "Execution Context, Application" on page 421.

Command

Throughout this chapter, the term "command" refers to a variety of AutoCAD constructs. A command consists of a program sequence performed as a logical unit of work that can be requested by the user or one of the AutoCAD scripting engines. No matter what construct is used, a command can be undone independently of other actions performed during system operation.

Specifically for the MDI API, a command is a sequence of code that begins by locking a document and ends by unlocking a document. In common cases, this locking and unlocking will be performed by ObjectARX but during other times the application must do the locking and unlocking directly. All of the following AutoCAD constructs are commands:

- AutoCAD built-in commands.
- Built-in commands executed directly from the command processor, such as F2 for change screen. This includes function and control keys.
- AutoLISP function invocations, which can be defined either in AutoLISP or in an ObjectARX application using `acedDefun()`.
- External program commands defined in *acad.pgp*.
- AcEd-registered commands registered from AutoCAD.
- Actions taken from a modeless dialog window or some other external process, typically hosted by an ObjectARX application.
- A set of actions taken from an ActiveX application in an external process.
- Actions taken from VBA through the ActiveX interface.
- Right-click context menu invocations.

Command, Multi-Document

A set of commands that appears as one command to the user, during which the user can change the current document and a continued logical flow of user prompting is maintained. For example, if an active command is prompting for user input in the current document and the user switches the document, the application cancels the command in the old current document, and queues up a command to commence execution in the new current document.

Command, Nonreentrant

A command that cannot be executed in more than one document at a time. Nonreentrancy can be used for commands that should not be available for more than one document at a time, or when the demands of supporting multiple instantiation are too great to be worth the overhead.

Command Processor

The standard input message polling mechanism in AutoCAD that facilitates combined keyboard and digitizer interaction. A separate command processor exists for each open document. The state of the command processor is maintained as an execution context.

NOTE Commands that execute outside the context of a single document, such as modeless dialogs and toolbars posted by AutoCAD or ObjectARX applications, execute from within the application context.

Current Document

Programmatic requests can be made to cause a document's execution context to become active without the user actually perceiving the document as "activated." This is a transient state only, used primarily by ActiveX and VBA applications.

Database

An AutoCAD drawing, specifically an instance of `AcDbDatabase`. Although the database is part of a document, it is not synonymous with a document.

Document

A document consists of an MDI document window, an execution context, an associated editor state, and a single current database, plus any number of side databases that are opened in association with it. The current database is the one being displayed and edited via commands. The side databases are either used by xref or for general use. The document also includes system variables that are associated with a given drawing such as the current viewport variable. Documents are uniquely identified by their address, which is the `AcApDocument*` pointer.

Drawing

Synonymous with database.

Edit Session

Usually synonymous with document, but sometimes includes its entire history since the document was opened, as well as the current state of the session.

Execution Context, Application

The command state that is active when new Windows messages are pending. It is independent from all document execution contexts. The following types of commands execute from this context:

- External ActiveX Automation requests (such as Visual Basic)
- VBA
- Modeless dialog boxes

These types of commands typically work on the active document, although they are not bound to do so. The intent is to handle document locking and unlocking reasonably transparently for external ActiveX applications and VBA. However, ObjectARX applications posting modeless dialogs will be required to lock and unlock documents explicitly in order to interact with their databases.

MDI-Aware

ObjectARX applications (and ActiveX and COM applications, but not necessarily Visual LISP applications) that meet all criteria needed to be successfully executed in an MDI AutoCAD. These criteria are listed in the section "MDI-Aware Level" on page 424. ObjectARX applications can register themselves as MDI-Aware by calling

```
acrxDynamicLinker->registerAppMDIAware(pkt);
```

when receiving the `kInitAppMsg` within their `acrxEntryPoint()` function.

Per-Application

A data structure that needs to exist only once per application.

Per-Context

A data structure that needs to be instantiated and maintained for each execution context, including document execution contexts and the application execution context. The AutoCAD command processor is an example of a per-context instantiation.

Per-Document

Any data structure, value, or other item that needs to be instantiated and maintained for each document.

Quiescent

When the command processor in a given edit session has no active AutoCAD commands, ObjectARX commands, Visual LISP evaluations, ActiveX requests, AutoCAD menu macros, or VBA macros. At this point, the Command prompt is being displayed in the command window. Notice that modeless dialogs and toolbars can be posted, as they do not operate through the command processor.

Session

Synonymous with application.

Undo Stack

A repository of state recorded during an edit session, which is used to undo the edit session command by command, when requested. Databases are often associated with an undo stack, which is supplied by the host application. In the case of AutoCAD, databases are typically opened under only one document at a time, because undo stacks correspond to documents.

SDI System Variable

ObjectARX provides a compatibility mode for the single drawing interface (SDI) of previous releases. It is controlled by the SDI system variable. Possible values for SDI are shown in the following table:

SDI system variable values

Value	Meaning
0	MDI is enabled
1	SDI mode set by user
2	SDI mode implied by loaded non–MDI-aware applications
3	SDI mode implied both by user and by loaded non–MDI-aware applications

Be aware of the following restrictions on changing the value of SDI:

- SDI can only be set to values of 2 or 3 by AutoCAD, as applications are loaded and unloaded.
- Always check the current value of SDI before making changes. If the current value is 2 or 3, do not change it.
- Changing SDI to 1 from 0 will fail when AutoCAD has more than one document open.

All document lock checking is disabled when AutoCAD is running in any of the SDI modes.

NOTE This compatibility mode and the SDI variable will be removed in the next full release of AutoCAD.

Levels of Compatibility

Your ObjectARX application can have one of four levels of compatibility with MDI:

- SDI-Only
- MDI-Aware
- MDI-Capable
- MDI-Enhanced

SDI-Only is the minimum requirement, but MDI-Capable compatibility is recommended.

The MDI supports an execution context per document and provides a facility for allowing a single execution context to be active when switching documents.

SDI-Only Level

This is the basic level of compatibility and is not sufficient for most robust applications. This level will not allow your application to run under MDI, but it should work without failing under SDI.

To create an SDI-Only application

1 Register your application as SDI-Only by calling `AcRxDynamicLinker::registerAppNotMDIAware()` in the `kInitAppMsg` handler of `acrxEntryPoint()`.

2 Ensure that your application can take an `AcRx::kUnloadAppMsg` immediately following a return from processing the `AcRx::kInitAppMsg`. This will occur if your application doesn't register as being MDI-Aware, and AutoCAD already has multiple documents open.

MDI-Aware Level

This level of compatibility provides the minimal requirements for an ObjectARX application to function in an MDI environment without failing. An ObjectARX application must meet the requirements of this level of compatibility before being able to legitimately set the SDI system variable to 0. The following list summarizes the minimum requirements.

- Applications cannot keep per-document data in global or static variables.
- If a command is executed from the application context, it must provide explicit document locking.
- AutoLISP-registered commands must be prepared to act within multiple AutoLISP states.
- Applications must register themselves as MDI-Aware during their `AcRx::kInitAppMsg` handler in `acrxEntryPoint()`.

Each requirement is described in detail in the following sections.

Per-Document Data

Applications cannot keep per-document data in global or static variables. This includes `AcDbDatabase` objects, `AcDbObject` objects, `AcDbObjectId` values, header variable values, document variable values, selection sets, and other document-specific information. Any occurrence of per-document data in global and static variables might be corrupted if your application is run in multiple concurrent edit sessions.

To avoid corruption of data, you can encapsulate command behavior and data into classes. An instance of the class can be instantiated for each call to the command. As the command acquires document-specific data, it keeps its own per-instance copies of that data.

Another solution is to encapsulate all of the global and static data into a structure or class. A copy of the data is instantiated for each document. A local pointer to the appropriate instance is set at each entry point in the application. The local pointer is then used to access the per-document data.

Use the `documentActivated()` reactor to switch between instances of the encapsulated data.

You can create the per-document data on an as-needed basis, or create it when the application is first loaded. If created on an as-needed basis, as the application's registered commands or reactors are called, the current document is determined and a query is made to get the document's data. If it is not found, it is created at that time.

To create the per-document data when the application is first loaded, use an `AcApDocumentIterator` in the `AcRx::kInitAppMsg` handler to get a list of all open documents. Then use `AcApDocManagerReactor::documentCreated()` to know when to create additional per-document data for documents opened after the application is loaded.

Whichever method is used to allocate the per-document data, the application must use the `AcApDocManagerReactor::documentToBeDestroyed()` reactor in order to know when to delete the data. Applications should also delete the remaining data during the `AcRx::kUnloadAppMsg` handler.

Explicit Document Locking

There are two types of execution contexts, application and document. All registered commands and reactor callbacks are executed within a document's execution context. Windows messages and callbacks, and some `acrxEntryPoint()` messages are executed within the application context.

Explicit locking is required only in the application execution context. Locking and unlocking is automatically handled for commands executing in the document context.

Any commands that need to work outside the active document must manually perform document locking using the following lock types.

- Read only
- Exclusive read
- Shared write
- Exclusive write

Locking in the application execution context can be done by calling `acDocManager->lockDocument()`. The following table describes the four levels of locking options:

Command lock types

Command Lock	Lock Mode	Command Flags	Description
Read only	(not locked)	ACRX_CMD_DOCREADLOCK	For read-only access to objects, locking is not necessary. For example, to open an AcDbObject for Acad::kForRead, or to call acedGetVar(), locking is not necessary.
Exclusive read	AcAp::kRead	ACRX_CMD_DOCREADLOCK ACRX_CMD_DOCEXCLUSIVELOCK	Using exclusive read mode prevents any other execution context from locking the document for write. This mode guarantees that the document will not be modified during the lock.
Shared write	AcAp::kWrite	(default)	The default lock mode. Multiple execution contexts can hold simultaneous shared write locks. A command can make changes to a document, and when the command is suspended, other commands can make changes to the document.
Exclusive write	AcAp::kXWrite	ACRX_CMD_DOCEXCLUSIVELOCK	Guarantees that your execution context has exclusive access to modify document resources.

AutoLISP Commands

AutoLISP commands must be aware that there is a separate AutoLISP stack for each document. This means that AutoLISP variables should be handled in the same way as other per-document global and static data. For more information, see "Per-Document Data" on page 424.

The `AcRx::kLoadDwgMsg` message in `acrxEntryPoint()` is sent for each document open when an application is first loaded, and when any new documents are opened while the application is running. The messages are sent from the corresponding document's execution context.

Registering as MDI-Aware

Applications that have met all of these criteria must register themselves as MDI-Aware in their `AcRx::kInitAppMsg` handler in `acrxEntryPoint()`, by using the function `acrxDynamicLinker->registerAppAsMDIAware()`. Applications not registered as MDI-Aware cannot be loaded when more than one document is open. If such an application is loaded, additional documents cannot be opened.

MDI-Capable Level

Reaching this level of compliance involves making your code work as efficiently and naturally in MDI mode as it does (or did) in SDI mode.

- Support document switching.

  Code sections comprising `AcEd`-registered commands that are invoked directly from those constructs will likely pause for user input, and are more likely susceptible to corruption when multiple sessions are being done. The most common case is to enter a command in one document, prompt for user input, then switch documents and enter the same command in a different document. As long as you are maintaining vital information on a per-open-document basis, your application should function properly. Otherwise, your code should disable document switching.
- Maintain good performance.

  When it becomes time to look at performance, a lot of heap-resident pointer dereferencing to what were formerly static addresses can bog down program performance. In this case, the alternative would be to maintain a static memory buffer of elements of the current document, which would be scanned in from and written out to the document-specific heap elements.

MDI-Enhanced Level

These additional steps will make your application integrate completely with the MDI.

- Consider migrating your `AcEditorReactor::commandXxx()` and `AcEditorReactor::LispXxx()` callbacks to work instead from `AcDocMananagerReactor::documentLockModeWillChange()` and `AcDocMananagerReactor::documentLockModeChanged()` notifications. Then they will account for application execution context operations that were previously difficult to detect by ObjectARX applications.

- Avoid using `acedCommand()`. Use `AcApDocManager::setStringToExecute()` instead, because it has a document parameter.
- Avoid using the `kLoadDwg` and `kUnloadDwg` cases, and use the `documentToBeCreated()` and `documentToBeDestroyed()` reactors instead.
- Support the document-independent database feature. For more information, see "Document-Independent Databases" on page 439.
- Support multi-document commands. For more information, see "Multi-Document Commands" on page 432.
- Support all events that occur within the application execution context. For more information, see "Application Execution Context" on page 436.

Interacting with Multiple Documents

This section describes the three main levels of interaction an ObjectARX application should have with multiple documents within a given command invocation. In the next three subsections, a "normal command" is either a built-in AutoCAD command, an AcEd-registered command, or an AutoLISP function, subject to restrictions on use of `acDocManager->curDocument()`.

Accessing the Current Document and Its Related Objects

The key call an ObjectARX application must make when it gains control is to find out the current document, which can be accomplished with the function `acDocManager->curDocument()`.

NOTE The current document is not always the active document. This is the case during transitional states, such as when the `documentToBeActivated()` reactor occurs. Do not attempt extensive processing during transitional states. Consider using `mdiActiveDocument()` if you are interested in the active document.

From the current document, you can determine the current database, the relevant transaction manager, and your application's associated document-specific state, and then do whatever needs to be done before returning.

Once a command has stored the current document and associated information on its stack, it does not need to query the current document again until completion. Whenever a prompt for user input is made, the user can switch

documents, but if that is done, the current command is suspended and its stack state is saved until the document is reactivated.

If your application is operating from the application execution context, it must lock and unlock the current document to modify anything associated with it. It can do so by directly invoking the `AcApDocManager::lockDocument()` and `unlockDocument()` member function.

If your application is operating from an ObjectARX or AutoLISP function, no locking should be necessary, as the system establishes the locks and removes them automatically around commands and AutoLISP expressions.

Accessing Databases Associated with Noncurrent Documents

Sometimes you will need to look at, or modify, databases associated with other documents, but not need any explicit user input in them. You might even need to look at something associated with another document.

To merely examine databases associated with other documents, you need not lock the document, although if the document is locked in `AcAp::kXWrite` mode by another execution context, you will be denied access to any of its elements.

To modify databases associated with other documents, or to prevent other execution contexts from modifying them for a period of time, you must lock the document, specifying `AcAp::kXWrite`, `AcAp::kWrite`, or `AcAp::kRead`, depending on your intent. If the document's command processor is not quiescent, it is usually already locked, and if it mutually excludes your lock, you will be denied access.

NOTE When modifying database objects in a noncurrent document, if you need to use transactions, be sure to use the transaction manager associated with the document. Such modifications will not be undoable from the current document. Instead they will be recorded with their host document's undo stack, and undone by using undo when the host document is current.

When finished with the information associated with a document, be sure to unlock it as soon as possible, to minimize the potential for conflicts with other commands.

Setting the Current Document without Activating It

There are several features that implicitly operate on the current document, which may be different from the active document. You can use the function `AcApDocManager::setCurDocument()` with the `activate` parameter set to `kFalse` to make the current document and active document different. The features for which this is necessary include:

- Using any user interaction functions, such as the `acedXXX()` functions.
- Creating a database to be associated with a particular document.
- Obtaining or manipulating a selection set without requiring user interaction.
- Using functions described in *aced.h*.

When the active and current documents are different, all user input functions and members concerning the document, such as the graphics screen, will be disabled. This includes functions for both ObjectARX and ActiveX applications.

Whenever you set the current document without also activating it, the `setCurDocument()` caller should restore the current document to be the same as the active document when finished. However, if this is not done by the time that the next input event is processed, the current document will be switched back to the active document.

Document Event Notification

ObjectARX applications must implement a document manager reactor to receive notification about document status changes if any of the following conditions apply:

- They need to manage per-document state.
- They need to be notified whenever a document or its database(s) are going to be modified, or are done being modified.
- They need to keep track of document switches, that is, which document is being made current or active.

The `AcApDocManReactor` makes callbacks when changes in document status take place, such as opening, closing, activation, deactivation, and changing the lock status of documents.

Application-Specific Document Objects

This section outlines how MDI-Aware applications need to be structured. Virtually all ObjectARX application developers must maintain a map between the system-supplied document objects and corresponding application-specific data. Any such map should be keyed with specific `AcApDocument` pointer values (addresses).

This requires that the application at least implement callbacks for the `AcApDocManagerReactor` methods `documentCreated()` and `documentToBeDestroyed()`, to create and delete the corresponding document-specific state. Make sure your `AcApDocument` pointers are up to date, as they will likely be reused as documents are terminated and created. As an alternative, you can implement handlers for when your `acrxEntryPoint()` function is invoked with `AcRx::kLoadDwgMsg` and `AcRx::kUnloadDwgMsg` messages, which are invoked with the document in question being current.

Such application-specific data should contain any state that must be associated with each open document that must persist across commands. One implementation alternative would be to maintain an `AcArray` template of a class whose instances consist of an `AcApDocument` pointer and a pointer to, or instance of, your document-specific state, and whose == operator is overloaded to compare only the `AcApDocument*` member. Another approach would be to maintain a pair of arrays with corresponding elements, do a find on the document pointers, and fetch the corresponding element out of the other array.

Nonreentrant Commands

Nonreentrant commands are those commands that cannot be invoked in one document when already in progress in a different document. This should be regarded as a way to make your application MDI-Aware quickly, without having to isolate all of your command-specific state information.

Making a Command Nonreentrant

To make a command nonreentrant

1. Declare a static Boolean variable in your application for each command you wish to be nonreentrant. Statically initialize each variable to FALSE.
2. Whenever a user enters the command or action you want to prevent reentrancy to, first check its static Boolean variable. If it is FALSE, set it to TRUE and continue the command. If it is TRUE, the command is being reentered, so post a message asking the user to complete the command in the other document that it is being used in.
3. Always set the Boolean variable back to FALSE when the command is completed, canceled, or terminates for any reason.

Nonreentrant AutoCAD Commands

Some AutoCAD commands are nonreentrant. These are commands that ask for user input and therefore would normally allow a document window switch, but would cause serious complications if it were switched, or if the command was called again in another document. Three commands fit this profile:

- TABLET
- NEW
- OPEN

The TABLET command is restricted because the user is defining how the pointing device is going to work, so that operation needs to be finished before anything else can be done. Because NEW and OPEN have to open a window, and then ask for a name (when FILEDIA=0), switching to another window at that moment could be error-prone.

Multi-Document Commands

A multi-document command allows users to switch documents at selected user prompts, while the command remains in control. The ability to have a single command remain active across documents is very complex. At the point of execution when a user has picked another document to switch to, all documents are polling for input. They therefore have potentially intricate command processor states established, including possibly nested commands, AutoLISP expressions, scripts active, and all in arbitrary nesting order.

One can't easily define, let alone graft, the pertinent elements of one state into another without major effort. Instead, the best strategy is for the application to retain control across user-specified document switches, even if the application has to be implemented as different commands in different windows. Then, all an application needs is a mechanism to chain separate commands running in different documents, and control which command to start when changing documents.

To synchronize the actions of the multiple commands, implement a reactor that overrides the following `AcApDocManager` reactor functions:

```
virtual void
documentActivated(
    AcApDocument* pActivatedDoc);

virtual void
documentToBeDeactivated(
    AcApDocument* pDeActivatedDoc);
```

The `documentToBeActivated()` reactor function can also be used, but it occurs before the document is activated. The document context has not been set in this case.

These callbacks are invoked whenever the user clicks on a different document to activate it. The reactors should only be used on `AcApDocManager` when in an initial command just before prompting for user input, at a point when document switching is to be supported. The reactor should be removed when any or all such prompts are completed or canceled. From the callback, invoke:

```
virtual Acad::ErrorStatus
sendStringToExecute(
    AcApDocument* pAcTargetDocument,
    const char * pszExecute,
    bool bActivate = true,
    bool bWrapUpInactiveDoc = false) = 0;
```

This function queues up a string to be interpreted the next time the specified document is activated. The string should typically be a command invocation (we'll call this the secondary command), but can also be an AutoLISP expression, a command fragment, or a menu token. The string limit is 296 bytes, so longer sequences should be implemented as a SCRIPT command running a temporary script, or as an AutoLISP expression to load and execute an AutoLISP program. The new document will be locked according to the new command's lock level as specified during its registration.

If the input prompt in the initial command looks the same as the first prompt in the secondary command, the user need not be aware that two separate commands are taking place.

NOTE Because this technique involves calling from the `documentActivated()` method, you should pass `kFalse` into the `bActivate` parameter to avoid errors or infinite loops.

Also, to manage the flow of control across documents, this callback should maintain whatever transition state the application needs. For example, a nonreentrant variant could remember the original document, and a flag for each document to indicate whether it is already active, and therefore not have to invoke `sendStringToExecute()`.

When a multi-document command completes, the controlling application should be sure the command left no pending command states in previous documents. The application can do this by sending an ESC or ENTER to the documents it has traversed through, by using the `bWrapUpInactiveDoc` parameter of `sendStringToExecute()`. If this is not done, documents may be left in a non-quiescent state.

Coordination between the initial command and the secondary command (and possibly multiple invocations thereof) must be managed through static or heap-resident variables.

Both the initial and secondary commands should be registered through `acedRegCmds()` or `acedDefun()`. The initial command should be coded to complete successfully on its own, in case the user decides to perform the entire command without switching documents. The second command need not be a full command, just a portion of a command that can be invoked to accumulate information (rooted in a static structure) from different open documents, and apply the results. The second command may also have to be coded such that it can be reentered in yet another document, if that is how the command is supposed to be structured.

Remember that UNDO runs separately in each document when designing these constructs.

NOTE The "normal" `acedSSGet()` is not viable, because it can prompt multiple times, thus not returning any selection set in progress. Instead, `acedEntSel()` is used, because it either returns an entity, or RTNONE, meaning the user is really done, or RTCAN, which can be either a real cancel or a "moved to another document" signal. Set the local "done" flag, perform the action, then queue up ESC to every other active document so that the command is finished up in that document the next time the user goes to click into it.

Disabling Document Switching

Commands that have long computation processes and poll for cancelation are susceptible to having user events or external ActiveX requests come in and cause problems with their internal state if they change documents. Therefore, some commands will disable document switching during their processing phase. Here is a list of commands built into AutoCAD or supplied with bundled ObjectARX applications that disable document switching while processing:

- PLOT
- REGEN
- RENDER
- HIDE
- SHADE

The following commands disable document switching throughout their invocation:

- NEW
- OPEN
- TABLET

In addition, `AcApDocManager::disableDocumentActivation()` and `AcApDocManager::enableDocumentActivation()` will disable and re-enable document activation. The prototypes for these methods are:

```
virtual Acad::ErrorStatus
disableDocumentActivation() = 0;
```

```
virtual Acad::ErrorStatus
enableDocumentActivation() = 0;
```

The following function indicates whether document activation is enabled:

```
virtual bool
isDocumentActivationEnabled() = 0;
```

Application Execution Context

The application execution context is distinct from the document execution context. The application execution context processes the Windows message loop.

Code Invoked under the Application Execution Context

The principal application code constructs that are normally invoked under the application execution context are:

- VBA-initiated ActiveX requests (implemented in ObjectARX).
- ActiveX requests made from external processes, including Visual Basic.
- Modeless dialog boxes posted by ObjectARX applications, or any DLL loaded by AutoCAD.
- All ObjectARX service calls made from the application context, including any ObjectARX-defined callouts that can be triggered from them, including custom object and entity virtual members, AcDb*, AcRx* reactor members, and `AcEditorReactor` members concerning database objects.

Code Differences under the Application Execution Context

There are a number of differences between code executing in the application execution context and code in other contexts. The differences are described below:

- It is not part of the command processor state of any specific document.
- It can activate different documents without immediately suspending itself, although it must complete and return before the new active document can process its input.
- Document switching is disabled when prompting for user input, either via ActiveX or ObjectARX user input requests.
- AutoLISP is also disabled when prompting for user input in this context.
- In the cases of modeless dialogs and external process-generated ActiveX requests, the code must lock the documents, including the current document. The use of the `IAcadDocument` methods `StartUndoMarker()` and `EndUndoMarker()` will apply a `kWriteLock` to the document.
- The command facility may not be used from the application execution context, specifically the `acedCommand()` and `acedCmd()` functions.

- The `AcApDocManager::sendStringToExecute()` and `AcApDocManager::activateDocument()` methods change the active document but do not suspend execution of the code running under the application context. They will suspend execution of code running in a document execution context. The `AcApDocManager::sendStringToExecute()` method will always queue the string when invoked from the application context, while invoking it from a document context will either queue the string if the activate parameter is `kFalse`, or immediately suspend the document context if the activate parameter is `kTrue`.

Other Application Execution Context Considerations

There are also certain capabilities and restrictions that apply to code executing in the application execution context.

- When the execution context your code is running under is not implicit in your code structure, you can make this query to find if it is the application execution context:
```
Adesk::Boolean
AcApDocManager::isApplicationContext() const;
```
- All ActiveX user input members may be used, but make sure that you are invoking them on the utility object associated with the active and current document. As noted above, document switching will be disabled when user input is obtained in this context. You can obtain the `IAcadDocument*` instance that corresponds to the current `AcApDocument` via the call:
```
acDocManager()->curDocument()->cDoc()->GetIDispatch(
    BOOL bAddRef);
```
- All ObjectARX user input functions may be called, with the current active document being implicitly used. As noted above, document switching will be disabled when user input is obtained in this context.
- Application code executing from the application context can use the following member function to switch either the current and active document, together or individually, as desired.
```
virtual Acad::ErrorStatus
setCurDocument(
    AcApDocument* pDoc,
    AcAp::DocLockMode = AcAp::kNone,
    bool activate = false) = 0;
```
- By alternating between prompting for user input and changing or activating the current document, one can prompt for input from multiple documents from a single execution context and a single sequence of code. The drawback is that document switching by the user is disabled when

prompting for input, so the code needs to know which document it wants to switch to.

- When the active and current documents differ, be aware that the ActiveX and ObjectARX user input functions will not operate properly. Use the `curDocument()` and `mdiActiveDocument()` functions to check the current and active documents.
- If the application leaves with the current document and active document different, the next input event will restore the current document back to the active document.
- When code executing from the application context is prompting for user input using the ActiveX user input functions, automatic interactive document switching is disabled, although current document switching can be performed.

Database Undo and Transaction Management Facilities

Most ObjectARX applications will not need to deal with undo and transaction management, but they should be aware of some important points:

- Undo and transaction management is performed on a per-document basis. In AutoCAD, it is controlled through (or in conjunction with) document locking.
- Whenever documents are locked for `kWrite` or `kXWrite`, a "begin command" undo bracket is written to the file, and then database and other modifications are performed. When the documents are unlocked from `kWrite` or `kXWrite` status, the corresponding "end command" undo bracket will be written. (Note that these can be nested.) By the time an application is finished operating on a document, it should have balanced its document lock and unlock requests. If the requests are not balanced, the undo file will work incorrectly, leaving actions out of sync from after the first write lock, and through the first lock balanced with an unlock. A subsequent undo request should put it back in sync.
- A parameter for establishing the command name is provided, which is displayed when an UNDO command is performed. The undo markers created are the same as for built-in AutoCAD and ObjectARX commands, and can therefore be managed via UNDO GROUP.
- Documents can have undo performed independently from each other.
- By default, when an instance of `AcDbDatabase` is created, its undo and transaction management is associated with the current document.

- Note that there are two methods of `AcEditorReactor` that are used to hook up databases with a document's undo facilities and transaction manager: `databaseConstructed()` and `databaseToBeDestroyed()`. If you receive such notification, be aware that the association between the database and any documents is undefined at that time, so document locking may or may not be required in the `databaseConstructed()` callback. Of course, any action that undoes any modifications done at that time will also undo the creation of the database.
- The default `AcDbDatabase` constructors will query the `AcDbHostApplicationServices` object for an undo controller.

Document-Independent Databases

To participate in undo in AutoCAD, databases must be associated with a document, because each document has an independent undo stack. However, this feature is in direct conflict with the need to load databases whose contents are intended to be shared across document edit sessions. In other words, you must decide between the following two scenarios for your side databases:

- Associate a database with a specific document, and don't allow edits to it from other edit sessions, and possibly load a DWG or DXF file into multiple databases for each edit session that needs it.
- Load a DWG or DXF file to share it across edit sessions, and have no automatic undo for it. Either don't support undo for them at all (it is fine if they are read-only, or only updated for actual saves, or are under revision control), or be very careful when using undo.

In ObjectARX, the former scenario is the default. Whenever a new instance of `AcDbDatabase` is instantiated, it is associated with the current document. This is one of the reasons an application needs to change the current document without activating the new document.

The `AcDbDatabase` class provides the following function, which disables the database undo and disassociates the database from the document:

```
void
disableUndoRecording(
    Adesk::Boolean disable);
```

Any AcDb reliance on any document-specific system variables will assume the built-in defaults for document-independent databases. Also, there is no need to lock any documents to access document-independent databases.

NOTE Developers who think about triggering an independent undo controller from multiple document undo controllers should remain aware that performing undo in a given document can lead to inconsistency and corruption. For example: Database X has an undo controller not associated with any document. Modifications from Document A are made to Database X, then modifications from Document B, which rely on objects created or modified from the Document A modifications. Now, undo is applied in Document A. The changes made to Document B will be corrupted.

An MDI-Aware Example Application

The following example shows a simple ObjectARX application that is MDI-Aware. This is the example code from "Using a Database Reactor" on page 399, with code added to make the application MDI-Aware. The new code is shown in boldface.

```
class AsdkDbReactor;

class AsdkDocReactor: public AcApDocManagerReactor
{
public:
    virtual void documentToBeActivated(AcApDocument* pDoc);
    virtual void documentCreated(AcApDocument* pDoc);
    virtual void documentToBeDestroyed(AcApDocument* pDoc);
};

class AsdkPerDocData
{
    friend class AsdkAppDocGlobals;
public:
    AsdkPerDocData(AcApDocument* pDoc);
private:
    AcApDocument* m_pDoc;
    AsdkPerDocData* m_pNext;
    long m_EntAcc;          // Entity count
    AsdkDbReactor* m_pDbr;// Pointer to database reactor
};
```

```
class AsdkAppDocGlobals
{
public:
    AsdkAppDocGlobals(AcApDocument* pDoc);
    void setGlobals(AcApDocument* pDoc);
    void removeDocGlobals(AcApDocument *pDoc);
    void removeAllDocGlobals(AsdkPerDocData* pTarget);
    void unload();
    long &entityCount();
    void incrementEntityCount();
    void decrementEntityCount();
    AsdkDbReactor *dbReactor();
    void setDbReactor(AsdkDbReactor *pDb);
private:
    AsdkPerDocData *m_pHead;
    AsdkPerDocData *m_pData;
    AsdkDocReactor *m_pDocReactor;
};

AsdkAppDocGlobals *gpAsdkAppDocGlobals;


// Custom AcDbDatabaseReactor class for Database
// event notification.
//
class AsdkDbReactor : public AcDbDatabaseReactor
{
public:

    virtual void objectAppended(const AcDbDatabase* dwg,
        const AcDbObject* dbObj);

    virtual void objectModified(const AcDbDatabase* dwg,
        const AcDbObject* dbObj);

    virtual void objectErased(const AcDbDatabase* dwg,
        const AcDbObject* dbObj, Adesk::Boolean pErased);
};

// This is called whenever an object is added to the database.
//
void
AsdkDbReactor::objectAppended(const AcDbDatabase* db,
    const AcDbObject* pObj)
{
    printDbEvent(pObj, "objectAppended");
    acutPrintf(" Db==%lx\n", (long) db);
    gpAsdkAppDocGlobals->incrementEntityCount();
    acutPrintf("Entity Count = %d\n",
        gpAsdkAppDocGlobals->entityCount());
}

// This is called whenever an object in the database is modified.
//
```

```
void
AsdkDbReactor::objectModified(const AcDbDatabase* db,
    const AcDbObject* pObj)
{
    printDbEvent(pObj, "objectModified");
    acutPrintf(" Db==%lx\n", (long) db);
}

// This is called whenever an object is erased from the database.
//
void
AsdkDbReactor::objectErased(const AcDbDatabase* db,
    const AcDbObject* pObj, Adesk::Boolean pErased)
{
    if (pErased)
    {
        printDbEvent(pObj, "objectErased");
        gpAsdkAppDocGlobals->decrementEntityCount();
    else
    {
        printDbEvent(pObj, "object(Un)erased");
        gpAsdkAppDocGlobals->incrementEntityCount();
    }
    acutPrintf(" Db==%lx\n", (long) db);
    acutPrintf("Entity Count = %d\n",
        gpAsdkAppDocGlobals->entityCount());
}


// Prints the message passed in by pEvent; then
// proceeds to call printObj() to print the information about
// the object that triggered the notification.
//
void
printDbEvent(const AcDbObject* pObj, const char* pEvent)
{
    acutPrintf("  Event: AcDbDatabaseReactor::%s ", pEvent);
    printObj(pObj);
}

// Prints out the basic information about the object pointed
// to by pObj.
//
void
printObj(const AcDbObject* pObj)
{
    if (pObj == NULL)
    {
        acutPrintf("(NULL)");
        return;
    }

    AcDbHandle objHand;
    char handbuf[17];
```

```
    // Get the handle as a string.
    //
    pObj->getAcDbHandle(objHand);
    objHand.getIntoAsciiBuffer(handbuf);

    acutPrintf(
        "\n   (class==%s, handle==%s, id==%lx, db==%lx)",
        pObj->isA()->name(), handbuf,
        pObj->objectId().asOldId(), pObj->database());
}

// Document swapping functions
//
void
AsdkDocReactor::documentToBeActivated(AcApDocument *pDoc)
{
    gpAsdkAppDocGlobals->setGlobals(pDoc);
}

void
AsdkDocReactor::documentCreated(AcApDocument *pDoc)
{
    gpAsdkAppDocGlobals->setGlobals(pDoc);
}

void
AsdkDocReactor::documentToBeDestroyed(AcApDocument *pDoc)
{
    gpAsdkAppDocGlobals->removeDocGlobals(pDoc);
}

AsdkPerDocData::AsdkPerDocData(AcApDocument *pDoc)
{
    m_pDoc = pDoc;
    m_pNext = NULL;
    m_EntAcc = 0;
    m_pDbr = NULL;
}

AsdkAppDocGlobals::AsdkAppDocGlobals(AcApDocument *pDoc)
{
    m_pData = m_pHead = NULL;
    m_pDocReactor = new AsdkDocReactor();
    acDocManager->addReactor(m_pDocReactor);
}

// Iterate through the list until the documents's global data is
// found. If it is not found, create a new set of document globals.
//
```

```
void
AsdkAppDocGlobals::setGlobals(AcApDocument *pDoc)
{
    AsdkPerDocData *p_data = m_pHead, *prev_data = m_pHead;
    while (p_data != NULL)
    {
        if (p_data->m_pDoc == pDoc)
        {
            m_pData = p_data;
            break;
        }
        prev_data = p_data;
        p_data = p_data->m_pNext;
    }
    if (p_data == NULL)
    {
        if (m_pHead == NULL)
            m_pHead = m_pData = new AsdkPerDocData(pDoc);
        else
           prev_data->m_pNext = m_pData = new AsdkPerDocData(pDoc);
    }
}

// Delete the globals associated with pDoc.
//
void
AsdkAppDocGlobals::removeDocGlobals(AcApDocument *pDoc)
{
    AsdkPerDocData  *p_data = m_pHead, *prev_data = m_pHead;
    while (p_data != NULL)
    {
        if (p_data->m_pDoc == pDoc)
        {
            if (p_data == m_pHead)
                m_pHead = p_data->m_pNext;
            else
                prev_data->m_pNext = p_data->m_pNext;
            if (m_pData == p_data)
                m_pData = m_pHead;
            delete p_data;
            break;
        }
        prev_data = p_data;
        p_data = p_data->m_pNext;
    }
}

// Delete all the doc globals in the list (recursively).
//
```

```
void
AsdkAppDocGlobals::removeAllDocGlobals(AsdkPerDocData *p_target)
{
    if (p_target == NULL)
        return;
    if (p_target->m_pNext != NULL)
        removeAllDocGlobals(p_target->m_pNext);
    if (p_target->m_pDbr != NULL)
    {
    acdbHostApplicationServices()->workingDatabase(
        )->removeReactor(p_target->m_pDbr);
    delete p_target->m_pDbr;
    p_target->m_pDbr = NULL;
    }
    delete p_target;
}

// Application was unloaded - delete everything associated with this
// document.
//
void AsdkAppDocGlobals::unload()
{
    acDocManager->removeReactor(m_pDocReactor);
    delete m_pDocReactor;
    removeAllDocGlobals(m_pHead);
    m_pHead = m_pData = NULL;
}

long &
AsdkAppDocGlobals::entityCount()
{
    return m_pData->m_EntAcc;
}

void
AsdkAppDocGlobals::incrementEntityCount()
{
    m_pData->m_EntAcc++;
}

void
AsdkAppDocGlobals::decrementEntityCount()
{
    m_pData->m_EntAcc--;
}

AsdkDbReactor *
AsdkAppDocGlobals::dbReactor()
{
    return m_pData->m_pDbr;
}
```

```
void
AsdkAppDocGlobals::setDbReactor(AsdkDbReactor *pDb)
{
    m_pData->m_pDbr = pDb;
}

// Adds a reactor to the database to monitor changes.
// This can be called multiple times without any ill
// effects because subsequent calls will be ignored.
//
void
watchDb()
{
    AsdkDbReactor *pDbr;
    if (gpAsdkAppDocGlobals->dbReactor() == NULL)
    {
        pDbr = new AsdkDbReactor();
        gpAsdkAppDocGlobals->setDbReactor(pDbr);
        acdbHostApplicationServices()->workingDatabase(
            )->addReactor(pDbr);
        acutPrintf(
            "  Added Database Reactor to "
           "acdbHostApplicationServices()->workingDatabase().\n");
    }
}

// Removes the database reactor.
//
void
clearReactors()
{
    AsdkDbReactor *pDbr;
    if ((pDbr = gpAsdkAppDocGlobals->dbReactor()) != NULL)
    {
        acdbHostApplicationServices()->workingDatabase(
            )->removeReactor(pDbr);
        delete pDbr;
        gpAsdkAppDocGlobals->setDbReactor(NULL);
    }
}
```

```
// ObjectARX entry point function
//
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxUnlockApplication(appId);
        acrxRegisterAppMDIAware(appId);
        gpAsdkAppDocGlobals = new AsdkAppDocGlobals(curDoc());
        gpAsdkAppDocGlobals->setGlobals(curDoc());
        acedRegCmds->addCommand("ASDK_NOTIFY_TEST",
            "ASDK_WATCH",
            "WATCH",
            ACRX_CMD_TRANSPARENT,
            watchDb);

        acedRegCmds->addCommand("ASDK_NOTIFY_TEST",
            "ASDK_CLEAR",
            "CLEAR",
            ACRX_CMD_TRANSPARENT,
            clearReactors);
        break;
    case AcRx::kUnloadAppMsg:
        if (gpAsdkAppDocGlobals != NULL)
        {
            gpAsdkAppDocGlobals->unload();
            delete gpAsdkAppDocGlobals;
            gpAsdkAppDocGlobals = NULL;
        }
        acedRegCmds->removeGroup("ASDK_NOTIFY_TEST");
        break;
    }
    return AcRx::kRetOK;
}
```

Transaction Management

17

This chapter describes the transaction model, which can be used to operate on AcDb objects. In this model, multiple operations on multiple objects are grouped together into one atomic operation called a transaction.

Transactions can be nested and can be ended or aborted at the discretion of the client. This model can be used in conjunction with the regular per-object open and close mechanism described in chapter 5, "Database Objects."

In This Chapter

- Overview of Transaction Management
- Transaction Manager
- Nesting Transactions
- Transaction Boundaries
- Obtaining Pointers to Objects in a Transaction
- Newly Created Objects and Transactions
- Commit-Time Guidelines
- Undo and Transactions
- Mixing the Transaction Model with the Open and Close Mechanism
- Transactions and Graphics Generation
- Transaction Reactors
- Example of Nested Transactions

Overview of Transaction Management

The transaction model encapsulates multiple operations on multiple objects by several clients as one atomic operation called a transaction. Inside a transaction boundary, clients can obtain object pointers from object IDs. Object pointers thus obtained are valid until the transaction is ended or aborted by the client. If the transaction is ended successfully, operations on the objects are committed. If the transaction is aborted, operations on the objects are canceled.

Operating on objects using this paradigm has several advantages over the regular open and close mechanism described in chapter 5, "Database Objects." The open and close mechanism is suitable for simple operations on a single object or a small group of objects. However, there are certain restrictions on opening objects this way. For example, if an object is open for read, you cannot open it for write at the same time. If an object is open for write, you cannot open it for write a second time. For a list of conflict errors associated with the open and close mechanism, see chapter 5, "Database Objects." The transaction model is more lenient, and obtaining object pointers from object IDs for a particular mode usually succeeds if the object is associated with a transaction.

Depending upon your application, there can be other disadvantages to using the open and close mechanism. If your application opens and closes the same object a number of times in the course of one operation—for example, a command—you'll incur serious inefficiencies due to these multiple opens and closes. A number of time-consuming operations are associated with closing an object. If you open an object for write, modify it, and then close it, the undo records of the modification are committed to the undo file, graphics for the object are generated, and notifications are fired. All these operations are performed every time the object is closed. If you transact your operation and obtain a pointer to the object using a transaction, all the activities mentioned above happen only once, at the end of the transaction. The result is improved efficiency and a smaller undo file, because the number of records going into the undo file is reduced.

Also, if you have a complex network where objects refer to each other by object ID, you want to be able to obtain a pointer to an object in any module of your program without worrying if another module or another application has that object opened. These activities are only possible using the transaction model because transactions group operations and allow obtaining pointers from object IDs across module boundaries.

Transaction Manager

The transaction manager is a global manager object, similar to the editor, that is in charge of maintaining transactions. It is an instance of `AcTransactionManager` and is maintained in the system registry. You can obtain it from the system registry using the macro `actrTransactionManager`, which expands to

```
#define actrTransactionManager  \
AcTransactionManager::cast(
    acrxSysRegistry()->at(AC_TRANSACTION_MANAGER_OBJ))
```

The transaction manager should be used to start, end, or abort transactions. It can also provide information such as the number of active transactions at any moment (see the following section, "Nesting Transactions") and a list of all the objects whose pointers have been obtained in all the transactions. The transaction manager maintains a list of reactors to notify clients of events such as the start, end, or abort of a transaction.

In addition to these managerial capabilities, the transaction manager can also be used to obtain pointers from object IDs. When this is done, the object is associated with the top (most recent) transaction. The transaction manager can also be used to queue up all the objects in all the transactions for graphics update and to flush the queue.

The transaction manager object is created and managed by the system. You should not delete it.

Nesting Transactions

Transactions can be nested—that is, you can start one transaction inside another and end or abort the recent transaction. The transaction manager maintains transactions in a stack, with the most recent transaction at the top of the stack. When you start a new transaction using `AcTransactionManager::startTransaction()`, the new transaction is added to the top of the stack and a pointer to it is returned (an instance of `AcTransaction`). When someone calls `AcTransactionManager::endTransaction()` or `AcTransactionManager::abortTransaction()`, the transaction at the top of the stack is ended or aborted.

When object pointers are obtained from object IDs, they are always associated with the most recent transaction. You can obtain the recent transaction using `AcTransactionManager::topTransaction()`, then use

`AcTransaction::getObject()` or `AcTransactionManager::getObject()` to obtain a pointer to an object. The transaction manager automatically associates the object pointers obtained with the recent transaction. You can use `AcTransaction::getObject()` only with the most recent transaction.

When nested transactions are started, the object pointers obtained in the outer containing transactions are also available for operation in the innermost transaction. If the recent transaction is aborted, all the operations done on all the objects (associated with either this transaction or the containing ones) since the beginning of the recent transaction are canceled and the objects are rolled back to the state at the beginning of the recent transaction. The object pointers obtained in the recent transaction cease to be valid once it's aborted.

If the innermost transaction is ended successfully by calling `AcTransactionManager::endTransaction()`, the objects whose pointers were obtained in this transaction become associated with the containing transaction and are available for operation. This process is continued until the outermost (first) transaction is ended, at which time modifications on all the objects are committed. If the outermost transaction is aborted, all the operations on all the objects are canceled and nothing is committed.

Transaction Boundaries

Because you, not the system, are in charge of starting, ending, or aborting transactions, it's important to be aware of transaction boundaries. A transaction boundary is the time between the start and end or abort of a transaction. It's recommended that you confine your boundary to the smallest possible scope. For example, if you start a transaction in a function, try to end or abort the transaction before you return from that function, because you may not have knowledge of the transaction outside of the function. You need not follow this rule if you maintain some kind of a global manager for your transaction activities, but you still are responsible for aborting or ending all the transactions you start.

Multiple applications can use transaction management for their work, and operations on objects are committed at the end of the outermost transaction. Therefore, an AutoCAD command boundary is as far as you can stretch the boundary of your transactions. When a command ends, there should not be any active transactions. If there are any active transactions (the transaction stack is not empty) when a command ends, AutoCAD will abort. As an exception, transactions can still be active when an `acedCommand()` or a transparent

command ends, but they should all be resolved when a main command ends and AutoCAD returns to the Command prompt.

It's generally a good idea to start a transaction when one of your functions is invoked as part of a command registered by you and end it when you return from that function. You can generalize it to all the commands in AutoCAD using the `AcEditorReactor::commandWillStart()` and `AcEditorReactor::commandEnded()` notifications, but there are certain commands that should not be transacted. The following commands should not be transacted:

- ARX
- DXFIN
- INSERT
- NEW
- OPEN
- PURGE
- QUIT
- RECOVER
- REDO
- SAVE
- SCRIPT
- U
- UNDO
- XREF

Obtaining Pointers to Objects in a Transaction

Both `AcTransactionManager::getObject()` and `AcTransaction::getObject()` can be used to obtain object pointers from object IDs. Pointers thus obtained are associated with the most recent transaction. Trying to obtain a pointer using any other transaction results in an error. Also, pointers thus obtained are valid until the transaction they are associated with, or one of the containing transactions, is aborted. When the outermost transaction ends, changes on all the valid pointers up to that moment are committed.

Both of the `getObject()` functions take an argument of the type `AcDb::OpenMode` and you can get an object pointer for read, write, or notify. All of these requests succeed except for one case: if the object is notifying and the request is to obtain a pointer for write (that is, with an intention of modifying it), an error (`eWasNotifying`) is returned. An object should not be modified while it is notifying others of its state.

If you use the `getObject()` function to obtain an object pointer, you should never call `close()` on that object pointer. Calling `close()` is valid only if you obtained the pointer using `acdbOpenObject()` or the object was newly created. For more information on when you can call `close()` on an object pointer, see the following sections, "Newly Created Objects and Transactions" and "Mixing the Transaction Model with the Open and Close Mechanism."

Newly Created Objects and Transactions

There are two ways to deal with newly created objects in a transaction management context.

The recommended approach is to `close()` the object after adding it to the database or the appropriate container and store the ID that is returned to you. Right after closing the object, which commits it to the database, you can use the `getObject()` function to obtain a new pointer for your operations. Even if you call `close()` on the object after adding it to the database, its creation will be undone if the containing transaction is aborted. See "Mixing the Transaction Model with the Open and Close Mechanism" on page 455.

The alternate approach is to add your newly created, in-memory object to the database or to the appropriate container, which in turn will add it to the database. Then add it to the most recent transaction using `AcTransactionManager::addNewlyCreatedDBRObject()` or `AcTransaction::addNewlyCreatedDBRObject`. Now that it's associated with a transaction, it will be committed or uncreated depending on whether the transactions end successfully or abort.

Commit-Time Guidelines

When the outermost transaction ends, the transaction manager fires an `endCalledOnOutermostTransaction()` notification (see "Transaction Reactors" on page 456) and begins the commit process, in which modifications on all the objects associated with the transaction are committed to the database. Each object is committed individually, one after another, until all of them are committed. During this operation, do not modify any of the objects involved in the commit process and do not start any new transactions. If you do so, AutoCAD will abort with the error message `eInProcessOfCommitting`.

You can modify individual objects after each has been committed, but it is recommended that you cache the IDs of the objects you want to modify and

wait until you receive the `transactionEnded()` notification signaling the end of all the transactions, then do the modifications.

Undo and Transactions

The transaction model uses AutoCAD's undo mechanism and `AcDbObject::cancel()` in implementing `AcTransactionManager::abortTransaction()`. This requires that you do not include any operation that uses AutoCAD's subcommand undo mechanism in a transaction. This will confuse `AcDbTransactionManager::abortTransaction()` and might produce unexpected results. Examples of operations that use the subcommand undo mechanism are the PEDIT and SPLINEDIT commands.

Mixing the Transaction Model with the Open and Close Mechanism

The transaction model coexists with the regular open and close mechanism described in chapter 5, "Database Objects." However, if you are using the transaction model, it is recommended that you do not mix it with the open and close mechanism. For example, if you obtained a pointer to an object using `AcTransaction::getObject()`, you should not call `close()` on the object pointer, which could cause unexpected results and may crash AutoCAD. However, you are free to open and close a particular object even if transactions are active. You can also instantiate new objects, add them to the database, and close them while transactions are active. The primary purpose of having the mixed model is to allow simultaneous execution of multiple applications where some use transaction management and others do not, but all of them are operating on the same objects.

Transactions and Graphics Generation

You can use `AcTransactionManager::queueForGraphicsFlush()` and `AcTransactionManager::flushGraphics()` to draw entities on demand even if they are associated with the transactions and some transactions are active, which would mean the modification on the entities is not committed to the database. `AcTransactionManager::queueForGraphicsFlush()` queues up all

the eligible entities associated with all the transactions for graphics update and `AcTransactionManager::flushGraphics()` draws them. You can also use `AcDbEntity::draw()` to draw an individual entity. This helps you see a particular entity on the screen without waiting until the end of the outermost transaction when all the modifications to all the entities are drawn. Use `AcTransactionManager::enableGraphicsFlush()` to enable or disable the drawing of entities. When a command ends, you relinquish control of graphics generation and it is automatically enabled.

Transaction Reactors

The transaction manager has a list of reactors through which it notifies clients of events relevant to the transaction model. Currently, there are four events that send notification:

```
virtual void
transactionStarted(
    int& numTransactions);

virtual void
transactionEnded(
    int& numTransactions);

virtual void
transactionAborted(
    int& numTransactions);

virtual void
endCalledOnOutermostTransaction(
    int& numTransactions);
```

The first three notifications are fired when any transaction, including nested ones, is started, ended, or aborted. You can use these notifications in conjunction with `AcTransactionManager::numActiveTransactions()` to determine the transaction that is relevant to the notification. For example, if a call to `AcTransactionManager::numActiveTransactions()` returns zero in your override of `AcTransactionReactor::transactionEnded()` or `AcTransactionReactor::transactionAborted()`, you know the outermost transaction is ending or aborting.

The `endCalledOnOutermostTransaction()` notification signals the beginning of the commit process of all the modifications done in all the transactions. You can use this callback to do any necessary cleanup work before commit begins.

The argument in all the notifications represents the number of transactions that are active plus the ones that have completed successfully. It doesn't include the transactions that were started and aborted.

Example of Nested Transactions

The following example includes three nested transactions. The sequence of events follows.

To create nested transactions

1. Create a polygon and post it to the database.
2. Start Transaction 1:
   - Select the polygon and obtain a pointer to it. Open it for read.
   - Create an extruded solid using the polygon.
   - Create a cylinder in the middle of the extended polygon.
3. Start Transaction 2: Subtract the cylinder from the extrusion (creates a hole in the middle of the solid).
4. Start Transaction 3:
   - Slice the shape in half along the *X/Z* plane and move it along the *X* axis so you can view the two pieces.
   - Abort the transaction? Answer *yes*.
5. Start Transaction 3 (again): Slice the shape in half along the *Y/Z* plane and move it along *Y*.
6. End Transaction 3.
7. End Transaction 2.

NOTE If you abort at this point, Transactions 2 and 3 are both canceled. If you abort a containing transaction, all the nested transactions are aborted, even if they were successfully ended.

---

8. End Transaction 1.

The following is the code for this example:

```
void
transactCommand()
{
    Adesk::Boolean interrupted;
    Acad::ErrorStatus es = Acad::eOk;
    AcDbObjectId savedCylinderId,savedExtrusionId;

    // Create a poly and post it to the database.
    //
```

```
acutPrintf("\nCreating a poly...Please supply the"
    " required input.");
if ((es = createAndPostPoly()) != Acad::eOk)
    return;

// Start a transaction.
//
AcTransaction *pTrans
    = actrTransactionManager->startTransaction();
assert(pTrans != NULL);
acutPrintf("\n\n######  Started transaction one."
    " ######\n");

// Select the poly and extrude it.
//
AcDbObject   *pObj = NULL;
AsdkPoly     *pPoly = NULL;
AcDb3dSolid  *pSolid = NULL;
AcDbObjectId  objId;
ads_name      ename;
ads_point     pickpt;

for (;;) {
    switch (acedEntSel("\nSelect a polygon: ",
        ename, pickpt))
    {
    case RTNORM:
        acdbGetObjectId(objId, ename);

        if ((es = pTrans->getObject(pObj, objId,
            AcDb::kForRead)) != Acad::eOk)
        {
            acutPrintf("\nFailed to obtain an object"
                " through transaction.");
            actrTransactionManager->abortTransaction();
            return;
        }

        assert(pObj != NULL);
        pPoly = AsdkPoly::cast(pObj);

        if (pPoly == NULL) {
            acutPrintf("\nNot a polygon. Try again");
            continue;
        }
        break;
    case RTNONE:
    case RTCAN:
        actrTransactionManager->abortTransaction();
        return;
    default:
        continue;
    }
    break;
}
```

```
// Now that a poly is created, convert it to a region
// and extrude it.
//
acutPrintf("\nWe will be extruding the poly.");
AcGePoint2d c2d = pPoly->center();
ads_point pt;
pt[0] = c2d[0]; pt[1] = c2d[1]; pt[2] = pPoly->elevation();
acdbEcs2Ucs(pt,pt,asDblArray(pPoly->normal()),Adesk::kFalse);
double height;

if (acedGetDist(pt, "\nEnter Extrusion height: ",
    &height) != RTNORM)
{
    actrTransactionManager->abortTransaction();
    return;
}

if ((es = extrudePoly(pPoly, height,savedExtrusionId)) !=
    Acad::eOk) {
    actrTransactionManager->abortTransaction();
    return;
}

// Create a cylinder at the center of the polygon of
// the same height as the extruded poly.
//
double radius = (pPoly->startPoint()
    - pPoly->center()).length() * 0.5;
pSolid = new AcDb3dSolid;
assert(pSolid != NULL);

pSolid->createFrustum(height, radius, radius, radius);
AcGeMatrix3d mat(AcGeMatrix3d::translation(
    pPoly->elevation()*pPoly->normal())*
    AcGeMatrix3d::planeToWorld(pPoly->normal()));
pSolid->transformBy(mat);

// Move it up again by half the height along
// the normal.
//
AcGeVector3d x(1, 0, 0), y(0, 1, 0), z(0, 0, 1);
AcGePoint3d  moveBy(pPoly->normal()[0] * height * 0.5,
                    pPoly->normal()[1] * height * 0.5,
                    pPoly->normal()[2] * height * 0.5);
mat.setCoordSystem(moveBy, x, y, z);
pSolid->transformBy(mat);
addToDb(pSolid, savedCylinderId);
actrTransactionManager
    ->addNewlyCreatedDBRObject(pSolid);
pSolid->draw();
acutPrintf("\nCreated a cylinder at the center of"
    " the poly.");
```

```
// Start another transaction. Ask the user to select
// the extruded solid followed by selecting the
// cylinder. Make a hole in the extruded solid by
// subtracting the cylinder from it.
//
pTrans = actrTransactionManager->startTransaction();
assert(pTrans != NULL);
acutPrintf("\n\n######  Started transaction two."
    " ######\n");
AcDb3dSolid *pExtrusion, *pCylinder;

if ((es = getASolid("\nSelect the extrusion: ", pTrans,
    AcDb::kForWrite, savedExtrusionId, pExtrusion))
    != Acad::eOk)
{
    actrTransactionManager->abortTransaction();
    actrTransactionManager->abortTransaction();
    return;
}

assert(pExtrusion != NULL);

if ((es = getASolid("\nSelect the cylinder: ", pTrans,
    AcDb::kForWrite, savedCylinderId, pCylinder))
    != Acad::eOk)
{
    actrTransactionManager->abortTransaction();
    actrTransactionManager->abortTransaction();
    return;
}

assert(pCylinder != NULL);
pExtrusion->booleanOper(AcDb::kBoolSubtract, pCylinder);
pExtrusion->draw();
acutPrintf("\nSubtracted the cylinder from the"
    " extrusion.");

// At this point, the cylinder is a NULL solid. Erase it.
//
assert(pCylinder->isNull());
pCylinder->erase();

// Start another transaction and slice the resulting
// solid into two halves.
//
pTrans = actrTransactionManager->startTransaction();
assert(pTrans != NULL);
acutPrintf("\n\n#####  Started transaction three."
    " ######\n");
```

```
AcGeVector3d vec, normal;
AcGePoint3d sp,center;
pPoly->getStartPoint(sp);
pPoly->getCenter(center);
vec = sp - center;
normal = pPoly->normal().crossProduct(vec);
normal.normalize();
AcGePlane sectionPlane(center, normal);
AcDb3dSolid *pOtherHalf = NULL;
pExtrusion->getSlice(sectionPlane, Adesk::kTrue,
    pOtherHalf);
assert(pOtherHalf != NULL);

// Move the other half three times the vector length
// along the vector.
//
moveBy.set(vec[0] * 3.0, vec[1] * 3.0, vec[2] * 3.0);
mat.setCoordSystem(moveBy, x, y, z);
pOtherHalf->transformBy(mat);
AcDbObjectId otherHalfId;
addToDb(pOtherHalf, otherHalfId);

actrTransactionManager
    ->addNewlyCreatedDBRObject(pOtherHalf);
pOtherHalf->draw();
pExtrusion->draw();
acutPrintf("\nSliced the resulting solid into half"
    " and moved one piece.");

// Now abort transaction three, to return to the hole in
// the extrusion.
//
Adesk::Boolean yes = Adesk::kTrue;
if (getYOrN("\nLet's abort transaction three, yes?"
    " [Y] : ", Adesk::kTrue, yes,interrupted) == Acad::eOk
    && yes == Adesk::kTrue)
{
    acutPrintf("\n\n$$$$$$  Aborting transaction"
        " three. $$$$$$\n");
    actrTransactionManager->abortTransaction();
    acutPrintf("\nBack to the un-sliced solid.");
    pExtrusion->draw();
    char option[256];
    acedGetKword("\nHit any key to continue.", option);
} else {
    acutPrintf("\n\n>>>>>>  Ending transaction three."
        " <<<<<<\n");
    actrTransactionManager->endTransaction();
}
```

```
// Start another transaction (three again). This time,
// slice the solid along a plane that is perpendicular
// to the plane we used last time: that is the slice
// we really wanted.
//
pTrans = actrTransactionManager->startTransaction();
assert(pTrans != NULL);
acutPrintf("\n\n#####  Started transaction three."
    " ######\n");
moveBy.set(normal[0] * 3.0, normal[1] * 3.0,
    normal[2] * 3.0);
normal = vec;
normal.normalize();
sectionPlane.set(center, normal);
pOtherHalf = NULL;

pExtrusion->getSlice(sectionPlane, Adesk::kTrue,
    pOtherHalf);
assert(pOtherHalf != NULL);
mat.setCoordSystem(moveBy, x, y, z);
pOtherHalf->transformBy(mat);
addToDb(pOtherHalf, otherHalfId);
actrTransactionManager
    ->addNewlyCreatedDBRObject(pOtherHalf);
pOtherHalf->draw();
pExtrusion->draw();

acutPrintf("\nSliced the resulting solid into half"
    " along a plane");
acutPrintf("\nperpendicular to the old one and moved"
    " one piece.");

// Now, give the user the option to end all the transactions.
//
yes = Adesk::kFalse;

if (getYOrN("\nAbort transaction three? <No> : ",
    Adesk::kFalse, yes,interrupted) == Acad::eOk
    && yes == Adesk::kTrue)
{
    acutPrintf("\n\n$$$$$$  Aborting transaction"
        " three. $$$$$$\n");
    actrTransactionManager->abortTransaction();
    acutPrintf("\nBack to the un-sliced solid.");
} else {
    acutPrintf("\n\n>>>>>>  Ending transaction three."
        " <<<<<<\n");
    actrTransactionManager->endTransaction();
}
```

```
    yes = Adesk::kFalse;

    if (getYOrN("\nAbort transaction two? <No> : ",
        Adesk::kFalse, yes,interrupted) == Acad::eOk
        && yes == Adesk::kTrue)
    {
        acutPrintf("\n\n$$$$$$  Aborting transaction two."
            " $$$$$$\n");
        actrTransactionManager->abortTransaction();
        acutPrintf("\nBack to separate extrusion and"
            " cylinder.");
    } else {
        acutPrintf("\n\n>>>>>>  Ending transaction two."
            " <<<<<<\n");
        actrTransactionManager->endTransaction();
    }

    yes = Adesk::kFalse;
    if (getYOrN("\nAbort transaction one? <No> : ",
        Adesk::kFalse, yes,interrupted) == Acad::eOk
        && yes == Adesk::kTrue)
    {
        acutPrintf("\n\n$$$$$$  Aborting transaction one."
            " $$$$$$\n");
        actrTransactionManager->abortTransaction();
        acutPrintf("\nBack to just the Poly.");
    } else {
        actrTransactionManager->endTransaction();
        acutPrintf("\n\n>>>>>>  Ending transaction one."
            " <<<<<<\n");
    }
}

static Acad::ErrorStatus
createAndPostPoly()
{
    int nSides = 0;
    while (nSides < 3) {
        acedInitGet(INP_NNEG, "");
        switch (acedGetInt("\nEnter number of sides: ",
            &nSides))
        {
        case RTNORM:
            if (nSides < 3)
               acutPrintf("\nNeed at least 3 sides.");
            break;
        default:
            return Acad::eInvalidInput;
        }
    }
```

```
    ads_point center, startPt, normal;
    if (acedGetPoint(NULL, "\nLocate center of polygon: ",
        center) != RTNORM)
    {
        return Acad::eInvalidInput;
    }

    startPt[0] = center[0];
    startPt[1] = center[1];
    startPt[2] = center[2];

    while (asPnt3d(startPt) == asPnt3d(center)) {
        switch (acedGetPoint(center,
            "\nLocate start point of polygon: ", startPt)) {
            case RTNORM:
                if (asPnt3d(center) == asPnt3d(startPt))
                    acutPrintf("\nPick a point different"
                        " from the center.");
                break;
            default:
                return Acad::eInvalidInput;
        }
    }

    // Set the normal to the plane of the polygon to be
    // the same as the Z direction of the current UCS,
    // (0, 0, 1) since we also got the center and
    // start point in the current UCS. (acedGetPoint()
    // returns in the current UCS.)
    normal[X] = 0.0;
    normal[Y] = 0.0;
    normal[Z] = 1.0;
    acdbUcs2Wcs(normal, normal, Adesk::kTrue);
    acdbUcs2Ecs(center, center, normal, Adesk::kFalse);
    acdbUcs2Ecs(startPt, startPt, normal, Adesk::kFalse);
    double elev = center[2];

    AcGePoint2d cen = asPnt2d(center),
                start = asPnt2d(startPt);
    AcGeVector3d norm = asVec3d(normal);
    AsdkPoly *pPoly = new AsdkPoly;
    if (pPoly==NULL)
        return Acad::eOutOfMemory;
    Acad::ErrorStatus es;

    if ((es=pPoly->set(cen, start, nSides, norm,
        "transactPoly",elev))!=Acad::eOk)
        return es;

    pPoly->setDatabaseDefaults(
        acdbHostApplicationServices()->workingDatabase());
    postToDb(pPoly);
    return Acad::eOk;
}
```

```
// Extrudes the poly to a given height.
//
static Acad::ErrorStatus
extrudePoly(AsdkPoly* pPoly, double height, AcDbObjectId&
savedExtrusionId)
{
    Acad::ErrorStatus es = Acad::eOk;

    // Explode to a set of lines.
    //
    AcDbVoidPtrArray lines;
    pPoly->explode(lines);

    // Create a region from the set of lines.
    //
    AcDbVoidPtrArray regions;
    AcDbRegion::createFromCurves(lines, regions);
    assert(regions.length() == 1);
    AcDbRegion *pRegion
        = AcDbRegion::cast((AcRxObject*)regions[0]);
    assert(pRegion != NULL);

    // Extrude the region to create a solid.
    //
    AcDb3dSolid *pSolid = new AcDb3dSolid;
    assert(pSolid != NULL);
    pSolid->extrude(pRegion, height, 0.0);

    for (int i = 0; i < lines.length(); i++) {
        delete (AcRxObject*)lines[i];
    }

    for (i = 0; i < regions.length(); i++) {
        delete (AcRxObject*)regions[i];
    }

    // Now we have a solid. Add it to database, then
    // associate the solid with a transaction. After
    // this, transaction management is in charge of
    // maintaining it.
    //
    pSolid->setPropertiesFrom(pPoly);
    addToDb(pSolid, savedExtrusionId);
    actrTransactionManager
        ->addNewlyCreatedDBRObject(pSolid);
    pSolid->draw();
    return Acad::eOk;
}
```

```
static Acad::ErrorStatus
getASolid(char*          prompt,
          AcTransaction* pTransaction,
          AcDb::OpenMode mode,
          AcDbObjectId   checkWithThisId,
          AcDb3dSolid*&  pSolid)
{
    AcDbObject   *pObj = NULL;
    AcDbObjectId  objId;
    ads_name      ename;
    ads_point     pickpt;

    for (;;) {
        switch (acedEntSel(prompt, ename, pickpt)) {
        case RTNORM:
            AOK(acdbGetObjectId(objId, ename));
            if (objId != checkWithThisId) {
                acutPrintf("\n Select the proper"
                    " solid.");
                continue;
            }
            AOK(pTransaction->getObject(pObj, objId, mode));
            assert(pObj != NULL);
            pSolid = AcDb3dSolid::cast(pObj);
            if (pSolid == NULL) {
                acutPrintf("\nNot a solid. Try again");
                AOK(pObj->close());
                continue;
            }
            break;
        case RTNONE:
        case RTCAN:
            return Acad::eInvalidInput;
        default:
            continue;
        }
        break;
    }
    return Acad::eOk;
}
```

Deep Cloning

18

In This Chapter

- Deep Clone Basics
- Implementing deepClone() for Custom Classes

This chapter describes the deep clone functions, which copy an object or any objects owned by the copied object. It covers both basic use of the `AcDbDatabase::deepCloneObjects()` function, as well as the more advanced topic of overriding the `deepClone()` and `wblockClone()` functions of the `AcDbObject` class. Editor reactor notification functions related to the deep clone, wblock clone, and insert operations are also discussed.

Deep Clone Basics

The deep clone functions copy an object and its ownership references. Any pointer references are ignored. The wblock clone function copies hard owners and hard pointers and ignores the soft references. In addition to copying this hierarchy of owned objects, both the deep clone functions and the wblock clone functions also handle the cloned object's references, translating the references to point to new objects if necessary.

To initiate a cloning operation, use one of the following functions:

```
AcDbDatabase::deepCloneObjects()
AcDbDatabase::wblock()
AcDbDatabase::insert()
```

`AcDbDatabase::deepCloneObjects()` only supports cloning within a single database. If you need to clone objects between databases, use either `wblock()`, `insert()`, or a combination of both (such as `wblock()` to a temporary database, and then `insert()` that database into an existing destination database).

When using `AcDbDatabase::insert()`, only insert to destination databases that have already been built. You can obtain a fully built (and possibly fully populated) destination database by using the current drawing to build a new database with a constructor parameter of `Adesk::kTrue` or by creating an empty new database using a constructor parameter of `Adesk::kFalse` and then calling `AcDbDatabase::readDwgFile()` on it to fill it in.

In general, to use `AcDbDatabase::deepCloneObjects()`, `AcDbDatabase::wblock()`, or `AcDbDatabase::insert()` functions in your code, you do not need to be aware of how the object ID map is filled in or exactly what happens during each stage of deep cloning. If you are creating a new class and you want to override the `AcDbObject::deepClone()` or `AcDbObject::wblockClone()` functions, you'll need to be familiar with the details of those functions, which are described in "Implementing deepClone() for Custom Classes" on page 476.

The `AcDbObject::deepClone()` and `AcDbObject::wblockClone()` functions should not be called directly on a custom object in application code. They are only called as part of a chain from a higher-level cloning operation.

Using clone() versus deepClone()

The `AcRxObject::clone()` function clones a single object. The `AcDbObject::deepClone()` function clones the object and any objects owned

by that object. The `deepClone()` function also translates the cloned object's references.

In general, the `deepClone()` function is the safer of the two functions. If you are cloning a simple object such as an ellipse, the two functions may be equivalent. But if you are cloning a complex object such as a polyline, the `clone()` function isn't adequate because it clones only the polyline object. With the `deepClone()` function, you clone the polyline as well as its vertices.

Key Concepts of Cloning

This section describes some of the key terms and concepts used throughout this discussion of deep clone and wblock clone: filing, ownership, the ID map, and the cloning and translating steps.

Cloning and Filing

The deep clone and wblock clone operations both use object filing to copy (clone) an object. A new object is created, which will be the clone. Next, the original object is filed out to memory using `dwgOut()`. Finally, the data is filed in to the new cloned object using `dwgIn()`.

Cloning and Ownership

Relationships between objects are stored in the object as a data member of `AcDbObjectId` class. There are four different types of relationships between objects—hard owners, soft owners, hard pointers, and soft pointers. For example, if you create an entity that requires a text style, that entity would have a data member of class `AcDbObjectId`, which would refer to an `AcDbTextStyleTableRecord`, and it would be filed out as a hard pointer ID. The way you file out the `AcDbObjectId` determines how the deep clone and wblock clone operations use the object ID. For more information, see "Object References" on page 310. Deep clone follows hard and soft owner connections, and wblock clone follows hard owner and pointer connections, as shown in the following figure:

Hard Owner deep clone wblock clone	Hard Pointer wblock clone
Soft Owner deep clone	Soft Pointer

Cloning and ID Map

The ID map is the mechanism for keeping track of a clone operation. The map consists of pairs of object IDs—the ID of the source object (referred to as the "key ID") and the ID of the cloned, or destination, object (referred to as the "value ID"). The ID map also contains additional ID pairs of noncloned objects that are needed for ID translation (see "Translation Phase" on page 477).

When `deepCloneObjects()` is called on certain objects, additional objects are cloned because of their ownership connection to the primary set of cloned objects. You can look at the ID map to see what additional objects were cloned.

Cloning and Translating

The deep clone and wblock clone operations actually consist of two steps: cloning and translating. The cloning step is where `dwgOut()` and `dwgIn()` are called and the objects are copied. The second step is the translation step, which uses the ID map to relink all the objects to reflect the new relationships.

During translation, all four types of object IDs have to be translated. Some objects have been cloned and are in the ID map, while others have not been cloned and are not in the map. During ID translation, if an ID pair corresponding to the reference is not found in the ID map, one of two things happens. If the reference is in the same database as the object that is referring to it, it is left alone. Otherwise, it is set to `NULL`. See "Translation Phase" on page 477.

Typical Deep Clone Operation

The following code excerpt shows a typical use of `AcDbDatabase::deepCloneObjects()`.

To initiate a deep clone operation

1. Obtain the set of objects to be cloned.
2. Put the object IDs into a list (of type `AcDbObjectIdArray`).
3. Create a new ID map (of class `AcDbIdMapping`) that will be filled in by the `deepCloneObjects()` function.
4. Call the `deepCloneObjects()` function, passing in the list of objects to be cloned, the owner object ID to which the cloned objects should be appended, and the ID map created in step 1.

In this example, the owner object ID is the model space block table record. The `deepCloneObjects()` function fills in the object ID map (`idMap`). The application can then iterate through the objects contained in the map using a special iterator object (`AcDbIdMappingIter`) and perform additional operations on those objects, such as transforming each object by a certain matrix.

The following code shows a typical use of `deepCloneObjects()`:

```
void
cloneSameOwnerObjects()
{
    // Step 1:  Obtain the set of objects to be cloned.
    ads_name sset;

    if (acedSSGet(NULL, NULL, NULL, NULL, sset) != RTNORM) {
        acutPrintf("\nNothing selected");
        return;
    }

    // Step 2: Add obtained object IDs to list of objects
    // to be cloned.
    long length;
    acedSSLength(sset, &length);
    AcDbObjectIdArray  objList;
    AcDbObjectId ownerId = AcDbObjectId::kNull;

    for (int i = 0; i < length; i++) {
        ads_name ent;
        acedSSName(sset, i, ent);
        AcDbObjectId objId;
        acdbGetObjectId(objId, ent);

        // Check to be sure this has the same owner as the first
        // object.
        //
        AcDbObject *pObj;
        acdbOpenObject(pObj, objId, AcDb::kForRead);

        if (pObj->ownerId() == ownerId)
            objList.append(objId);
        else if (i == 0) {
            ownerId = pObj->ownerId();
            objList.append(objId);
        }
        pObj->close();
    }

    acedSSFree(sset);

    // Step 3: Get the object ID of the desired owner for
    // the cloned objects.  We'll use model space for
    // this example.
    //
```

```
    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    AcDbObjectId  modelSpaceId;
    pBlockTable->getAt(ACDB_MODEL_SPACE, modelSpaceId);
    pBlockTable->close();

    // Step 4:  Create a new ID map.
    //
    AcDbIdMapping idMap;

    // Step 5: Call deepCloneObjects().
    //
    acdbHostApplicationServices()->workingDatabase()
        ->deepCloneObjects(objList, modelSpaceId, idMap);

    // Now we can go through the ID map and do whatever we'd
    // like to the original and/or clone objects.
    //
    // For this example, we'll print out the object IDs of
    // the new objects resulting from the cloning process.
    //
    AcDbIdMappingIter iter(idMap);
    for (iter.start(); !iter.done(); iter.next()) {
        AcDbIdPair idPair;
        iter.getMap(idPair);
        if (!idPair.isCloned())
            continue;
        acutPrintf("\nObjectId is: %Ld",
            idPair.value().asOldId());
    }
}
```

Cloning Objects from Different Owners

If you are cloning a set of objects from different owners, you'll need to divide the set of object IDs into separate groups for each owner. (The cloned objects and their owners should belong to the same database.) In the example in this section, the model space objects to be cloned are added to `objListMS`, and the paper space objects to be cloned are added to `objListPS`:

```
objListMS.append(objId);

objListPS.append(objId);
```

The `deepCloneObjects()` function is then called twice, using the same ID map. It is necessary to do all the cloning using a single ID map for the reference translation to be done properly. On the first call, the `deferXlation` parameter is set to `kTrue`. On the second (the last) call to `deepCloneObjects()`, `deferXlation` defaults to `kFalse`:

```
acdbHostApplicationServices()->workingDatabase()->
    deepCloneObjects(mslist, modelSpaceId, idMap, Adesk::kTrue);

acdbHostApplicationServices()->workingDatabase()->
    deepCloneObjects(pslist, paperSpaceId, idMap);
```

At this point, cloning concludes and all the references are translated. The following code deep clones objects belonging to different owners:

```
void
cloneDiffOwnerObjects()
{
    // Step 1: Obtain the set of objects to be cloned.
    // For the two owners we'll use model space and
    // paper space, so we must perform two acedSSGet() calls.
    // calls.
    //
    acutPrintf("\nSelect entities to be cloned to"
        " Model Space");

    ads_name ssetMS;
    acedSSGet(NULL, NULL, NULL, NULL, ssetMS);
    long lengthMS;
    acedSSLength(ssetMS, &lengthMS);
    acutPrintf("\nSelect entities to be cloned to"
        " Paper Space");

    ads_name ssetPS;
    if (acedSSGet(NULL, NULL, NULL, NULL, ssetPS)
        != RTNORM && lengthMS == 0)
    {
        acutPrintf("\nNothing selected");
        return;
    }

    long lengthPS;
    acedSSLength(ssetPS, &lengthPS);

    // Step 2: Add obtained object IDs to the lists of
    // objects to be cloned: one list for objects to
    // be owned by model space and one for those to
    // be owned by paper space.
```

```
AcDbObjectId ownerId = AcDbObjectId::kNull;

// For model space
//
AcDbObjectIdArray  objListMS;

for (int i = 0; i < lengthMS; i++) {
    ads_name ent;
    acedSSName(ssetMS, i, ent);
    AcDbObjectId objId;
    acdbGetObjectId(objId, ent);

    // Check to be sure this has the same owner as the first
    // object.
    //
    AcDbObject *pObj;
    acdbOpenObject(pObj, objId, AcDb::kForRead);

    if (pObj->ownerId() == ownerId)
        objListMS.append(objId);
    else if (i == 0) {
        ownerId = pObj->ownerId();
        objListMS.append(objId);
    }
    pObj->close();
}

acedSSFree(ssetMS);

// For paper space
//
ownerId = AcDbObjectId::kNull;
AcDbObjectIdArray  objListPS;

for (i = 0; i < lengthPS; i++) {
    ads_name ent;
    acedSSName(ssetPS, i, ent);
    AcDbObjectId objId;
    acdbGetObjectId(objId, ent);

    // Check to be sure this has the same owner as the first
    // object.
    //
    AcDbObject *pObj;
    acdbOpenObject(pObj, objId, AcDb::kForRead);

    if (pObj->ownerId() == ownerId)
        objListPS.append(objId);
    else if (i == 0) {
        ownerId = pObj->ownerId();
        objListPS.append(objId);
    }

    pObj->close();
}
```

```
    acedSSFree(ssetPS);

    // Step 3: Get the object ID of the desired owners for
    // the cloned objects.  We're using model space and
    // paper space for this example.
    //
    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);

    AcDbObjectId  modelSpaceId, paperSpaceId;
    pBlockTable->getAt(ACDB_MODEL_SPACE, modelSpaceId);
    pBlockTable->getAt(ACDB_PAPER_SPACE, paperSpaceId);
    pBlockTable->close();

    // Step 4:  Create a new ID map.
    //
    AcDbIdMapping idMap;

    // Step 5: Call deepCloneObjects().
    //
    acdbHostApplicationServices()->workingDatabase()
        ->deepCloneObjects(objListMS, modelSpaceId, idMap,
        Adesk::kTrue);
    acdbHostApplicationServices()->workingDatabase()
        ->deepCloneObjects(objListPS, paperSpaceId, idMap);

    // Now we can go through the ID map and do whatever we'd
    // like to the original and/or clone objects.
    //
    // For this example we'll print out the object IDs of
    // the new objects resulting from the cloning process.
    //
    AcDbIdMappingIter iter(idMap);
    for (iter.start(); !iter.done(); iter.next()) {
        AcDbIdPair idPair;
        iter.getMap(idPair);
        if (!idPair.isCloned())
            continue;
        acutPrintf("\nObjectId is: %Ld",
            idPair.value().asOldId());
    }
}
```

Implementing deepClone() for Custom Classes

The first part of this chapter described basic use of the `deepCloneObjects()` function. This section describes the behind-the-scenes details of deep cloning, so you can override the `deepClone()` and `wblockClone()` functions for your own custom objects and entities.

When overriding these functions for custom objects, you may wish to clone other objects with the custom object. These are referred to as dependent references. The custom object's `wblockClone()` function may call `wblockClone()` on the dependent reference, but must not call `deepClone()`. Likewise, the custom object's `deepClone()` function may call `deepClone()` on the dependent reference, but must not call `wblockClone()`. When a cloning request is forwarded in this fashion, you must also forward the `AcDbIdMapping` instance that was received in the call to the custom object's cloning function to the dependent object's cloning function. Never create a new instance of `AcDbIdMapping` to pass to the cloning function of the dependent reference.

AutoCAD Commands That Use Deep Clone and Wblock Clone

A number of AutoCAD commands use the `deepClone()` function to create new objects from old ones. In some cases, one version of the command performs deep cloning, while another version does not. Commands using `deepClone()` and `wblockClone()` are as follows:

COPY	Uses `deepClone()`.
ARRAY	Uses `deepClone()`.
MIRROR	If both the original and the mirrored objects are preserved, uses `deepClone()`. If the original objects are deleted, `deepClone()` is not used (the original objects are only mirrored).
BLOCK	Uses `deepClone()`. This command copies the entities into another space and erases the original entities.
INSERT	When you insert a drawing, this command uses `deepClone()` to copy the entities into the drawing.

WBLOCK	Uses `wblockClone()`. This function follows hard ownership and hard pointer connections only. All other copy commands that use `deepClone()` follow both hard and soft ownership connections from the primary object.
XREF BIND XBIND	Uses `wblockClone()` to bring the referenced entities into your current drawing.
EXPLODE	When you explode an object into its parts, no cloning is performed. When you explode a block reference, AutoCAD deletes the block reference entity and copies the individual entities into the drawing. This version of EXPLODE uses `deepClone()`.

Cloning Phase

During the cloning phase, when you call `deepClone()` on an object, AutoCAD checks to see if the cloned object (the primary object) owns any other objects. If it does, it calls `deepClone()` on those objects as well. This process continues until all owned objects have been cloned. Both hard and soft ownership connections are followed.

When you call `wblockClone()` on an object, AutoCAD follows hard owner and hard pointer connections and calls `wblockClone()` on those objects as well.

Translation Phase

For both the deep clone and wblock clone functions, objects that are referenced by the primary object are also translated. After the objects are copied, AutoCAD translates the references as described in the following three cases.

- *Case 1:* If the referenced object has been copied, the old reference is translated to refer to the copied object. In this case, it does not matter if the copied object is in the same database as the source objects or in a new database.
- *Case 2:* This case assumes that the source object and the copied object reside in the same database. If the referenced object has not been copied, the reference is left in place.
- *Case 3:* This case assumes that the source object and the copied object are in different databases. If the referenced object has not been copied, the reference to it is set to `NULL` (because it isn't in the destination database).

Translation Phase: Case 1

As an example of Case 1, suppose you have Entity A_1, which contains a pointer reference to Entity B_1. Both Entity A_1 and Entity B_1 are selected to be cloned. Before translation, Entity A_2 still refers to Entity B_1. After translation, Entity A_2 is updated to refer to Entity B_2.

Translation Phase: Case 2

As an example of Case 2, suppose you have the same two entities: Entity A_1 contains a pointer reference to Entity B_1. Entity A_1 is cloned, but Entity B_1 is not. The source and destination (cloned) objects are in the same database.

Translation Phase: Case 3

Case 3 is similar to Case 2, except the cloned objects are in a new database. In this case, the pointer reference of Entity A_2 is set to `NULL`, because Entity B_1 is not in the new database.

Named Object Dictionary

The named object dictionary has soft ownership of its entries. The entries are thus not cloned by `wblockClone()`. It is up to the application to copy those objects if necessary.

During the INSERT command, application-defined entries in the named object dictionary are not copied. The application must perform the desired cloning during the `beginDeepCloneXlation()` phase, adding the object IDs to the ID map and adding a new entry to the destination named object dictionary. For more information on `beginDeepCloneXlation()`, see "Editor Reactor Notification Functions" on page 504.

During the WBLOCK command, all IDs in the original named object dictionary are brought over to the destination named object dictionary, but the objects pointed to are not automatically copied. If the object an ID points to is not cloned by the application, the ID is removed from the destination dictionary during `endDeepClone()` translation. Again, the application needs to clone the objects during `beginDeepCloneXlation` and add the IDs to the

ID map. It does not need to add a new ID to the destination named object dictionary because this task was performed automatically.

The following example shows how you might write an `AcEditorReactor::beginDeepCloneXlation()` function for a user-defined dictionary of objects that is placed in the named object dictionary. The example refers only to the `kDcWblock` and `kDcInsert` contexts.

```
// This example demonstrates a way to handle objects in
// the named object dictionary for WBLOCK and INSERT.
// Our object is an AcDbDictionary, which is called
// "AsdkDictionary" in the named objects dictionary,
// containing our custom objects.
//
const char *kpDictionary = "AsdkDictionary";

// AsdkNODEdReactor is derived from AcEditorReactor.
//
void
AsdkNODEdReactor::beginDeepCloneXlation(
    AcDbIdMapping& idMap,
    Acad::ErrorStatus* pRetStat)
{
    Acad::ErrorStatus es;
    AcDbObjectId dictId;
    if (   idMap.deepCloneContext() != AcDb::kDcWblock
        && idMap.deepCloneContext() != AcDb::kDcInsert)
        return;

    // Get the "from" and "to" databases.
    //
    AcDbDatabase *pFrom, *pTo;
    idMap.origDb(pFrom);
    idMap.destDb(pTo);

    // See if the "from" database has our dictionary, and
    // open it.  If it doesn't have one, we are done.
    //
    AcDbDictionary *pSrcNamedObjDict;
    pFrom->getNamedObjectsDictionary(pSrcNamedObjDict,
                                     AcDb::kForRead);
    es = pSrcNamedObjDict->getAt(kpDictionary, dictId);
    pSrcNamedObjDict->close();

    if (es == Acad::eKeyNotFound)
        return;

    AcDbDictionary *pSrcDict;
    acdbOpenObject(pSrcDict, dictId, AcDb::kForRead);
    AcDbObject *pClone;
```

```
switch (idMap.deepCloneContext()) {
case AcDb::kDcWblock:
    // WBLOCK clones all or part of a drawing into a
    // newly created drawing.  This means that the
    // named object dictionary is always cloned, and
    // its AcDbObjectIds are in flux.  Therefore, you
    // cannot use getAt() or setAt() on the dictionary
    // in the new database.  This is because the
    // cloned dictionary references all refer to the
    // original objects.  During deep clone translation,
    // all cloned entries will be translated to the
    // new objects, and entries not cloned will be
    // "removed" by getting "translated" to NULL.
    //
    // The cloning of entries in our own dictionary are
    // not handled here. If all are to be cloned, then
    // call setTreatElementsAsHard(Adesk::kTrue) on the
    // dictionary. Otherwise, only those entries that
    // are referred to by hard references in other
    // wblocked objects will have been cloned via
    // those references.
    // In this example, we will always write out all of
    // the records.  Since TreatElementsAsHard is not
    // currently persistent, we reset it here each time.
    //
    pSrcDict->upgradeOpen();
    pSrcDict->setTreatElementsAsHard(Adesk::kTrue);
    pClone = NULL;
    pSrcDict->wblockClone(pTo, pClone, idMap,
                          Adesk::kFalse);
    if (pClone != NULL)
        pClone->close();
    break;
case AcDb::kDcInsert:
    // In INSERT, an entire drawing is cloned, and
    // "merged" into a pre-existing drawing.  This
    // means that the destination drawing may already
    // have our dictionary, in which case we have to
    // merge our entries into the destination
    // dictionary. First we must find out if
    // the destination named objects dictionary contains
    // our dictionary.
    //
    AcDbDictionary *pDestNamedDict;
    pTo->getNamedObjectsDictionary(pDestNamedDict,
                                   AcDb::kForWrite);

    // Since INSERT does not clone the destination
    // named object dictionary, we can use getAt()
    // on it.
    //
    es = pDestNamedDict->getAt(kpDictionary, dictId);
```

```
// If our dictionary does not yet exist in the
// named object dictionary, which is not itself
// cloned, we have to both clone and add our
// dictionary to it.  Since dictionary entries are
// ownership references, all of our entries will
// also be cloned at this point, so we are done.
//
if (es == Acad::eKeyNotFound) {
    pClone = NULL;
    pSrcDict->deepClone(pDestNamedDict,
                        pClone, idMap);

    // Unless we have overridden the deepClone()
    // of our dictionary, we should expect it to
    // always be cloned here.
    //
    if (pClone == NULL) {
        *pRetStat = Acad::eNullObjectId;
        break;
    }

    pDestNamedDict->setAt(kpDictionary,
                          pClone, dictId);
    pDestNamedDict->close();
    pClone->close();
    break;
}

pDestNamedDict->close();

// Our dictionary already exists in the destination
// database, so now we must "merge" the entries
// into it.  Since we have not cloned our
// destination dictionary, its object IDs are not in
// flux, and we can use getAt() and setAt() on it.
//
AcDbDictionary *pDestDict;
acdbOpenObject(pDestDict, dictId, AcDb::kForWrite);
AcDbObject *pObj, *pObjClone;
AcDbDictionaryIterator* pIter;
pIter = pSrcDict->newIterator();

for (; !pIter->done(); pIter->next()) {
    const char *pName = pIter->name();
    pIter->getObject(pObj, AcDb::kForRead);

    // If the dictionary contains any references
    // and/or other objects have references to it,
    // you must either use deepClone() or put the
    // ID pairs into the ID map here, so that they
    // will be in the map for translation.
    //
    pObjClone = NULL;
    pObj->deepClone(pDestDict, pObjClone, idMap);
```

```
            // INSERT usually uses a method of cloning
            // called CheapClone, where it "moves" objects
            // into the destination database instead of
            // actually cloning them.  When this happens,
            // pObj and pObjClone are pointers to the
            // same object.  We only want to close pObj
            // here if it really is a different object.
            //
            if (pObj != pObjClone)
                pObj->close();

            if (pObjClone == NULL)
                continue;

            // If the name already exists in our
            // destination dictionary, it must be changed
            // to something unique.  In this example, the
            // name is changed to an anonymous entry.
            // The setAt() method will automatically append
            // a unique identifier to each name beginning
            // with "*", for example: "*S04".
            //
            if (   pDestDict->getAt(pName, dictId)
                == Acad::eKeyNotFound)
                pDestDict->setAt(pName, pObjClone, dictId);
            else
                pDestDict->setAt("*S", pObjClone, dictId);

            pObjClone->close();
        }

        delete pIter;
        pDestDict->close();
        break;
    default:
        break;
    }
    pSrcDict->close();
}
```

Overriding the deepClone() Function

The sample code in this section is an approximation of the default behavior of `deepClone()`. The deep clone operation has two main stages:

- Cloning (you can override this stage)
- Translation (you will not need to reimplement this stage; it can be controlled by what is put into the ID map)

During the cloning stage in this example, information about the old object is copied to the new object using a specific type of filer to write out the object

and read it back. The filer keeps track of objects owned by the primary object so that they can be copied as well.

To complete the cloning stage

1. Create a new object of the same type as the old one.
2. Append the new object to its owner.

   - If the object is an entity, its owner is a block table record and you can use the `appendAcDbEntity()` function.
   - If the object is an `AcDbObject`, its owner is an `AcDbDictionary` and you can use the `setAt()` function to add it to the dictionary.

   If this is not a primary object, you would normally add it to the database using `addAcDbObject()` and then identify its owner using `setOwnerId()`. To establish ownership, the owner must file out the ID of the owned object using the appropriate ownership type.
3. Call `dwgOut()` on the original object, using a deep clone filer (`AcDbDeepCloneFiler`) to write out the object. (Or, if you are overriding the `wblockClone()` function, use an `AcDbWblockCloneFiler`.)
4. Rewind the filer and then call `dwgIn()` on the new object.
5. Call `setObjectIdsInFlux()` on each new object before you add its value to the object ID map. This important step is used to indicate that the newly created object is part of a deep clone operation and its object ID is subject to change as part of the translation stage. This flag is automatically turned off when translation is complete.
6. Add the new information to the `idMap`. The `idMap` contains `AcDbIdPairs`, which are pairs of old (original) and new (cloned) object IDs. The constructor for the ID pair sets the original object ID and the `isPrimary` flag. At this point, you set the object ID for the cloned object, set the `isCloned` flag to `TRUE`, and add (`assign`) it to the `idMap`.
7. Clone the owned objects. (This step is recursive.)

   - Ask the filer if there are any more owned objects. (For wblock clone, ask if there are any more hard objects.)
   - To clone a subobject, obtain its ID and open the object for read.
   - Call `deepClone()` on the object. (Note that `isPrimary` is set to `FALSE`, because this is an owned object.) The `deepClone()` function clones the object and sets its owner. It also adds a record to the ID map.
   - Close the subobject if it was created at this time.

The following sample code illustrates these steps:

```
Acad::ErrorStatus
AsdkPoly::deepClone(
    AcDbObject*    pOwner,
    AcDbObject*&   pClonedObject,
    AcDbIdMapping& idMap,
    Adesk::Boolean isPrimary) const
{
    // You should always pass back pClonedObject == NULL
    // if, for any reason, you do not actually clone it
    // during this call.  The caller should pass it in
    // as NULL, but to be safe, we set it here as well.
    //
    pClonedObject = NULL;

    // If this object is in the idMap and is already
    // cloned, then return.
    //
    bool isPrim = false;
    if (isPrimary)
        isPrim = true;

    AcDbIdPair idPair(objectId(), (AcDbObjectId)NULL,
                      false, isPrim);

    if (idMap.compute(idPair) && (idPair.value() != NULL))
        return Acad::eOk;

    // Step 1: Create the clone.
    //
    AsdkPoly *pClone = (AsdkPoly*)isA()->create();
    if (pClone != NULL)
        pClonedObject = pClone;    // Set the return value.
    else
        return Acad::eOutOfMemory;

    // Step 2: Append the clone to its new owner. In this
    // example, the original object is derived from AcDbEntity,
    // so the owner is expected to be an AcDbBlockTableRecord,
    // unless an ownership relationship is set up with another
    // object. In that case, it is important to establish a
    // connection to that owner. This sample shows a generic
    // method using setOwnerId().
    //
    AcDbBlockTableRecord *pBTR =
        AcDbBlockTableRecord::cast(pOwner);

    if (pBTR != NULL) {
        pBTR->appendAcDbEntity(pClone);
    } else {
        if (isPrimary)
            return Acad::eInvalidOwnerObject;
```

```
    // Some form of this code is only necessary if
    // anyone has set up an ownership for the object
    // other than with an AcDbBlockTableRecord.
    //
    pOwner->database()->addAcDbObject(pClone);
    pClone->setOwnerId(pOwner->objectId());
}

// Step 3: Now contents are copied to the clone. This is done
// using an AcDbDeepCloneFiler. This filer keeps a list of all
// AcDbHardOwnershipIds and AcDbSoftOwnershipIds contained in
// the object and its derived classes. This list is then used
// to know what additional, "owned" objects need to be cloned
// below.
//
AcDbDeepCloneFiler filer;
dwgOut(&filer);

// Step 4: Rewind the filer and read the data into the clone.
//
filer.seek(0L, AcDb::kSeekFromStart);
pClone->dwgIn(&filer);

// Step 5: This must be called for all newly created objects
// in deepClone(). It is turned off by endDeepClone()
// after it has translated the references to their
// new values.
//
pClone->setAcDbObjectIdsInFlux();

// Step 6: Add the new information to the ID map.  We can use
// the ID pair started above.
//
idPair.setValue(pClonedObject->objectId());
idPair.setIsCloned(Adesk::kTrue);
idMap.assign(idPair);

// Step 7: Using the filer list created above, find and clone
// any owned objects.
//
AcDbObjectId id;

while (filer.getNextOwnedObject(id)) {
    AcDbObject *pSubObject;
    AcDbObject *pClonedSubObject;

    // Some object's references may be set to NULL,
    // so don't try to clone them.
    //
    if (id == NULL)
        continue;
```

```
        // Open the object and clone it. Note that "isPrimary" is
        // set to kFalse here because the object is being cloned,
        // not as part of the primary set, but because it is owned
        // by something in the primary set.
        //
        acdbOpenAcDbObject(pSubObject, id, AcDb::kForRead);
        pClonedSubObject = NULL;
        pSubObject->deepClone(pClonedObject,
                              pClonedSubObject,
                              idMap, Adesk::kFalse);

        // If this is a kDcInsert context, the objects
        // may be "cheap" cloned. In this case, they are
        // "moved" instead of cloned. The result is that
        // pSubObject and pClonedSubObject will point to
        // the same object. Therefore, we only want to close
        // pSubObject if it really is a different object
        // than its clone.
        //
        if (pSubObject != pClonedSubObject)
            pSubObject->close();

        // The pSubObject may either already have been
        // cloned, or for some reason has chosen not to be
        // cloned.  In that case, the returned pointer will
        // be NULL.  Otherwise, since we have no immediate
        // use for it now, we can close the clone.
        //
        if (pClonedSubObject != NULL)
            pClonedSubObject->close();
    }

    // Leave pClonedObject open for the caller.
    //
    return Acad::eOk;
}
```

Overriding the wblockClone() Function

When a wblock clone operation is performed, AutoCAD constructs a valid database for the new drawing, which contains the named object dictionary, all the symbol tables, and the complete set of header variables. The following code approximates the default implementation of `wblockClone()`. The steps listed correspond to those listed in the section "Overriding the deepClone() Function" on page 484.

The `wblockClone()` function uses a filer of class `AcDbWblockCloneFiler`, which returns a list of the hard pointer and hard owner connections of the primary object. Before you call `wblockClone()` on these subobjects, you need

to check the owner of the subobject. At this point, you'll do one of two things:

- If you are the owner of the object, set the owner of the subobject to be the clone of yourself.
- If you are not the owner of the object, pass in the clone's database as the `pOwner` parameter in the `wBlockClone()` function call. At this time, the object is appended to the new database, receives an object ID, and is put into the ID map. The ID map entry for this object will specify FALSE for the `isOwnerTranslated` field.

If `pOwner` is set to the database, `wblockClone()` must set the owner of the cloned object to the same owner as that of the original object. Then, when the references are translated by AutoCAD, it will update the owner reference to the cloned object in the new database.

It is important to ensure that all owning objects are cloned. AutoCAD always clones the symbol tables, named object dictionary, model space, and paper space (for clone contexts other than `AcDb::kDcXrefBind`) during wblock clone. Applications with owning objects are responsible for ensuring that these objects are cloned if necessary. If an owning object is not cloned and not found in the ID map, wblock clone aborts with `AcDb::eOwnerNotSet`.

You must pass in the database as the owner of an object when you are copying an entity that references a symbol table record. For example, suppose you are calling `wblockClone()` on a sphere object. A block table record is the hard owner of this sphere object. The sphere object contains a hard reference to the layer table.

First, at the `beginDeepClone()` phase, the new database is created and set up with the default elements. The following figure shows the model space block table record and the layer table, because they're relevant to this topic. The cloning that occurs at this stage always happens during a wblock operation.

At the `beginWblock()` stage, the selection set is cloned, as shown in the following figure. In this example, the sphere is cloned.

Because the sphere contains a hard pointer to $Layer_1$, $Layer_1$ is cloned.

Next, pointers need to be translated to refer to the cloned objects, as shown in the following figure. The `beginDeepCloneXlation()` notification indicates the beginning of this stage.

The ID map for the previous figure at the time of `beginDeepCloneXlation()` is as follows:

ID map of the previous figure

KEY	VALUE	isCloned	isPrimary	isOwnerXlated
BTR1	BTR2	TRUE	FALSE	TRUE
SPH1	SPH2	TRUE	TRUE	TRUE
LT1	LT2	TRUE	FALSE	*
LTR1	LTR2	TRUE	FALSE	FALSE†

***** The layer table's owner is the database itself, so this entry is meaningless.

† During translation, this setting indicates that the layer will have its owner translated from LayerTable_1 to LayerTable_2.

The wblock clone process is used for xref bind as well as wblock. The needs of both are very similar, but there are a few differences that require special attention when overriding the `wblockClone()`.

Wblock clones all selected entities. However, xref bind never clones entities that are in paper space. This leaves two things to consider when creating objects or entities, and using `AcDbHardPointerIds`. First, at the beginning of any `AcDbEntity`'s `wblockClone()`, check to see if the cloning context is `AcDb::kDcXrefBind` and, if so, whether the entity is being cloned in paper space. If it is, then no cloning should be done and `wblockClone()` should return `Acad::eOk`.

If your custom class has any `AcDbHardPointerIds` that can point to entities (such as we do with `AcDbGroup`), then the entities might be in paper space and will therefore not get cloned. In that event, the `AcDbHardPointerIds` will be set to `NULL`.

Wblock does not follow hard pointer references across databases. However, xref bind does this all the time. For example, an entity in an xref drawing can be on a VISRETAIN layer in the host drawing. So, if you implement your `wblockClone()` with a loop to check for subobjects, and the subobject's

database is not the same as that of the object being cloned, you must skip the subobject if the cloning context is not `AcDb::kDcXrefBind`. For example:

```
if(pSubObject->database() != database()
    && idMap.deepCloneContext() != AcDb::kDcXrefBind)
{
    pSubObject->close();
    continue;
}
```

The following code shows overriding `wblockClone()`to implement it for a custom entity (`AsdkPoly`). This function is invoked with the code shown in "Editor Reactor Notification Functions" on page 504.

```
Acad::ErrorStatus
AsdkPoly::wblockClone(AcRxObject*    pOwner,
                      AcDbObject*&   pClonedObject,
                      AcDbIdMapping& idMap,
                      Adesk::Boolean isPrimary) const
{
    // You should always pass back pClonedObject == NULL
    // if, for any reason, you do not actually clone it
    // during this call.  The caller should pass it in
    // as NULL, but to be safe, it is set here as well.
    //
    pClonedObject = NULL;

    // If this is a fast wblock operation, no cloning
    // should take place, so we simply call the base class's
    // wblockClone() and return whatever it returns.
    //
    // For fast wblock, the source and destination databases
    // are the same, so we can use that as the test to see
    // if a fast wblock is in progress.
    //
    AcDbDatabase *pDest, *pOrig;
    idMap.destDb(pDest);
    idMap.origDb(pOrig);
    if (pDest == pOrig)
        return AcDbCurve::wblockClone(pOwner, pClonedObject,
            idMap, isPrimary);
```

```
// If this is an xref bind operation and this AsdkPoly
// entity is in paper space, then we don't want to
// clone because xref bind doesn't support cloning
// entities in paper space. Simply return Acad::eOk.
//
static AcDbObjectId pspace = AcDbObjectId::kNull;

if (pspace == AcDbObjectId::kNull) {
    AcDbBlockTable *pTable;
    database()->getSymbolTable(pTable, AcDb::kForRead);
    pTable->getAt(ACDB_PAPER_SPACE, pspace);
    pTable->close();
}

if (   idMap.deepCloneContext() == AcDb::kDcXrefBind
    && ownerId() == pspace)
    return Acad::eOk;

// If this object is in the idMap and is already
// cloned, then return.
//
bool isPrim = false;
if (isPrimary)
    isPrim = true;
AcDbIdPair idPair(objectId(), (AcDbObjectId)NULL,
                  false, isPrim);
if (idMap.compute(idPair) && (idPair.value() != NULL))
    return Acad::eOk;

// The owner object can be either an AcDbObject or an
// AcDbDatabase. AcDbDatabase is used if the caller is
// not the owner of the object being cloned (because it
// is being cloned as part of an AcDbHardPointerId
// reference).  In this case, the correct ownership
// will be set during reference translation. If
// the owner is an AcDbDatabase, then pOwn will be left
// NULL here, and is used as a "flag" later.
//
AcDbObject   *pOwn = AcDbObject::cast(pOwner);
AcDbDatabase *pDb = AcDbDatabase::cast(pOwner);

if (pDb == NULL)
    pDb = pOwn->database();

// Step 1: Create the clone.
//
AsdkPoly *pClone = (AsdkPoly*)isA()->create();
if (pClone != NULL)
    pClonedObject = pClone;    // Set the return value.
else
    return Acad::eOutOfMemory;
```

```
// Step 2: If the owner is an AcDbBlockTableRecord, go ahead
// and append the clone.  If not, but we know who the
// owner is, set the clone's ownerId to it. Otherwise,
// we set the clone's ownerId to our own ownerId (in
// other words, the original ownerId). This ID will
// then be used later, in reference translation, as
// a key to finding who the new owner should be.  This
// means that the original owner must also be cloned at
// some point during the wblock operation.
// EndDeepClone's reference translation aborts if the
// owner is not found in the ID map.
//
// The most common situation where this happens is
// AcDbEntity references to symbol table records, such
// as the layer an entity is on. This is when you will
// have to pass in the destination database as the owner
// of the layer table record.  Since all symbol tables
// are always cloned in wblock, you do not need to make
// sure that symbol table record owners are cloned.
//
// However, if the owner is one of your own classes,
// then it is up to you to make sure that it gets
// cloned.  This is probably easiest to do at the end
// of this function. Otherwise you may have problems
// with recursion when the owner, in turn, attempts
// to clone this object as one of its subobjects.
//
AcDbBlockTableRecord *pBTR = NULL;

if (pOwn != NULL)
    pBTR = AcDbBlockTableRecord::cast(pOwn);
if (pBTR != NULL) {
    pBTR->appendAcDbEntity(pClone);
} else {
    pDb->addAcDbObject(pClonedObject);
    pClone->setOwnerId(  (pOwn != NULL)
                       ? pOwn->objectId() : ownerId());
}

// Step 3: The AcDbWblockCloneFiler makes a list of
// AcDbHardOwnershipIds and AcDbHardPointerIds. These
// are the references that must be cloned during a
// wblock operation.
//
AcDbWblockCloneFiler filer;
dwgOut(&filer);

// Step 4: Rewind the filer and read the data into the clone.
//
filer.seek(0L, AcDb::kSeekFromStart);
pClone->dwgIn(&filer);
```

```
// Step 5:
// This must be called for all newly created objects
// in wblockClone. It is turned off by endDeepClone()
// after it has translated the references to their
// new values.
//
pClone->setAcDbObjectIdsInFlux();

// Step 6: Add the new information to the ID map. We can use
// the ID pair started above. We must also let the
// idMap entry know whether the clone's owner is
// correct, or needs to be translated later.
//
idPair.setIsOwnerXlated((Adesk::Boolean)(pOwn != NULL));
idPair.setValue(pClonedObject->objectId());
idPair.setIsCloned(Adesk::kTrue);
idMap.assign(idPair);

// Step 7: Using the filer list created above, find and clone
// any hard references.
//
AcDbObjectId id;

while (filer.getNextHardObject(id)) {
    AcDbObject *pSubObject;
    AcDbObject *pClonedSubObject;

    // Some object references may be set to NULL,
    // so don't try to clone them.
    //
    if (id == NULL)
        continue;

    // If the referenced object is from a different
    // database, such as an xref, do not clone it.
    //
    acdbOpenAcDbObject(pSubObject, id, AcDb::kForRead);

    if (pSubObject->database() != database()) {
        pSubObject->close();
        continue;
    }

    // To find out if this is an AcDbHardPointerId
    // versus an AcDbHardOwnershipId, check if we are the
    // owner of the pSubObject. If we are not,
    // then we cannot pass our clone in as the owner
    // for the pSubObject's clone.  In that case, we
    // pass in our clone's database (the destination
    // database).
    //
    // Note that "isPrimary" is set to kFalse here
    // because the object is being cloned, not as part
    // of the primary set, but because it is owned by
    // something in the primary set.
    //
```

```
        pClonedSubObject = NULL;
        if (pSubObject->ownerId() == objectId()) {
            pSubObject->wblockClone(pClone,
                                    pClonedSubObject,
                                    idMap, Adesk::kFalse);
        } else {
            pSubObject->wblockClone(
                pClone->database(),
                pClonedSubObject,
                idMap, Adesk::kFalse);
        }

        pSubObject->close();

        // The pSubObject may either already have been
        // cloned, or for some reason has chosen not to be
        // cloned.  In that case, the returned pointer will
        // be NULL.  Otherwise, since there is no immediate
        // use for it now, the clone can be closed.
        //
        if (pClonedSubObject != NULL)
            pClonedSubObject->close();
    }
    // Leave pClonedObject open for the caller.
    //
    return Acad::eOk;
}
```

NOTE Remember that when the `wblock()` function is in the process of executing, the pointer references in the destination database have not yet been translated. The following code does not work correctly, because although it tries to open the model space block table record of the destination database, the model space block table record of the source database is opened instead. The untranslated reference in the destination database's block table is still referring to the model space of the source database.

```
void
AsdkWblockReactor::otherWblock(
    AcDbDatabase*  pDestDb,
    AcDbIdMapping& idMap,
    AcDbDatabase*  pSrcDb)
{
    AcDbBlockTable       *pDestBlockTable;
    AcDbBlockTableRecord *pDestBTR;

    pDestDb->getSymbolTable(pDestBlockTable, AcDb::kForRead);
    pDestBlockTable->getAt(ACDB_MODEL_SPACE,
        pDestBTR, AcDb::kForRead);

    pDestBlockTable->close();
```

```
    // Now pDestBTR is pointing to pSrcDb database's model
    // space, not to the destination database's model space!
    // The code above is not correct!
}
```

To find the destination model space, you must look it up in the ID map:

```
void
AsdkWblockReactor::otherWblock(
    AcDbDatabase*  pDestDb,
    AcDbIdMapping& idMap,
    AcDbDatabase*  pSrcDb)
{
    // To find the destination model space, you must look
    // it up in the ID map:
    AcDbBlockTable *pSrcBlockTable;
    pSrcDb->getSymbolTable(pSrcBlockTable, AcDb::kForRead);

    AcDbObjectId srcModelSpaceId;
    pSrcBlockTable->getAt(ACDB_MODEL_SPACE,
        srcModelSpaceId);
    pSrcBlockTable->close();

    AcDbIdPair idPair;
    idPair.setKey(srcModelSpaceId);
    idMap.compute(idPair);
    AcDbBlockTableRecord *pDestBTR;

    acdbOpenAcDbObject((AcDbObject*&)pDestBTR,
        idPair.value(), AcDb::kForRead, Adesk::kTrue);
}
```

Using appendAcDbEntity() During Cloning

`AcDbBlockTableRecord::appendAcDbEntity()` requires valid `AcDbObjectIds` in order to do the append properly. During cloning, an entity may be appended to an `AcDbBlockTableRecord` only if `AcDbBlockTableRecord::isObjectIdsInFlux()` returns `Adesk::kFalse`. This indicates that the `AcDbBlockTableRecord` itself is not currently being cloned. The one exception to this rule occurs when the cloned `AcDbBlockTableRecord` is empty. Because an empty `AcDbBlockTableRecord` contains no untranslated `AcDbObjectIds`, the append will work properly. This situation arises during some forms of `wblock()`, and is described in more detail shortly.

If deep cloning is being called on individual entities, then their clones must be appended to the destination `AcDbBlockTableRecord`. However, when the `AcDbBlockTableRecord` itself is being deep cloned, then all its entities are cloned with it, and a call to `AcDbBlockTableRecord::appendAcDbEntity()` will not only be unnecessary, but would corrupt the cloned `AcDbBlockTableRecord`.

Default implementations of `deepClone()` and `wblockClone()` know when to call `AcDbBlockTableRecord::appendAcDbEntity()` by checking the `isPrimary` value. When an entity is being deep cloned by itself, `isPrimary` is true, and append is called. If the entity is being cloned as the result of a deep cloning of an `AcDbBlockTableRecord`, then `isPrimary` is false, and append is not called.

Normally, applications do not need to be concerned with this detail and can rely on the default implementation of `deepClone()` and `wblockClone()` to handle entities. However, situations can arise when applications may want to add entities during cloning, or use hard references to entities. The hard referenced entity will have an `isPrimary` value of `Adesk::kFalse` and will not call append, even when it may need to do so. This situation is covered in the next section.

The following examples and rules illustrate important aspects of cloning.

deepClone()

The `AcDbHardPointerId` reference problem mentioned above will not occur in this case because `deepClone()` does not follow `AcDbHardPointerId` references for cloning.

An application may create problems during `deepClone()` if it attempts to add new entities while the `AcDbObjectIds` are still in flux. Therefore, never attempt to call `AcDbBlockTableRecord::appendAcDbEntity()` on any cloned, user-defined `AcDbBlockTableRecords` until after the `AcEditorReactor::endDeepClone()` notification has been given. In contrast, you may safely append to the model space and paper space `AcDbBlockTableRecords`, because these are never cloned in `deepClone()`. Never try to add vertices to cloned `AcDb2dPolylines`, `AcDb3dPolylines`, `AcDbPolyFaceMeshes`, or `AcDbPolygonMeshes`, attributes to cloned `AcDbBlockReferences`, or entries to cloned dictionaries, until after the `AcEditorReactor::endDeepClone()` notification.

If you must create the entities during the cloning, then you will need to keep them in memory, along with their future owner's ID, until after the `AcEditorReactor::endDeepClone()` notification. They can be safely appended once the deep clone is completed.

wblockClone()

There are three versions of `AcDbDatabase::wblock()`:

1. WBLOCK*

```
Acad::ErrorStatus
AcDbDatabase::wblock(
    AcDbDatabase*& pOutputDatabase)
```

2. WBLOCK of a user-defined block

```
Acad::ErrorStatus
AcDbDatabase::wblock(
    AcDbDatabase*& pOutputDatabase,
    AcDbObjectId nObjId)
```

3. WBLOCK of a selection set

```
Acad::ErrorStatus
AcDbDatabase::wblock(
    AcDbDatabase*& pOutputDatabase,
    const AcDbObjectIdArray& pIdSet,
    const AcGePoint3d& pPoint3d)
```

One of the main internal differences between these three versions of wblock is their treatment of the model space and paper space `AcDbBlockTableRecords`. Because the entire database is being cloned in version one, all the entities in model space and paper space are cloned along with their containing paper and model space `AcDbBlockTableRecords`. However, in versions two and three, the intent is to clone only a selected set of entities. Although the model space and paper space `AcDbBlockTableRecords` are processed, these use a "shallow clone," which does *not* in turn clone all the entities contained in model space and paper space.

Even though the model space and paper space blocks have been cloned in versions two and three, they are empty. Therefore, it is not only acceptable to call `AcDbBlockTableRecord::AppendAcDbEntity()` to place the cloned entities into them, it is necessary to do so. (This is the exception to using `AcDbBlocKTableRecord::AppendAcDbEntity()` on `AcDbBlockTableRecords`, whose IDs are in flux). Also, in both versions two and three, the entities will have `isPrimary` set to `Adesk::kTrue` when they get their `wblockClone()` call. This is because the internal code individually clones the entities of the selection set, or the entities of the selected `AcDbBlockTableRecord`. It does not clone the `AcDbBlockTableRecord` itself. (Entities in nested blocks, however, will still have `isPrimary` set to `Adesk::kFalse`). This behavior is useful, as will be seen in the next section, in Case 1. It saves applications from having to know what type of WBLOCK operation is occurring.

Here are some more rules to keep in mind:

1. Never use `AcDbBlocKTableRecord::AppendAcDbEntity()` during `WBLOCK*`. If you must create new entities, you must keep them in memory, along with their future owner's ID, and then append them after `AcEdItorReactor::endDeepClone()`. This also applies to appending objects to `AcDbDictionaries`, polylines, polyfacemeshes, polygonmeshes, and block references.
2. In the other two forms of WBLOCK, only use `AcDbBlocKTableRecord::ApPendAcDbEntity()` when appending to model space or paper space. But with that exception, all the other restrictions mentioned for WBLOCK* still apply.

Handling Hard References to AcDbEntities During wblockClone()

If you create a custom object with either an `AcDbHardPointerId` or a hard-coded reference to an `AcDbEntity`, you are responsible for calling `AcDbBlockTableRecord::appendAcDbEntity()` on the referenced entity, when needed during `wblock()`. In this context, a hard-coded reference is any situation in which an object causes an entity to be included in a `wblockClone()` via some custom code written in an application.

It is necessary to do the appending manually because the default implementation of `AcDbDatabase::wblockClone()` will always set `isPrimary` to `Adesk::kFalse` when calling `wblockClone()` on referenced objects. If the object is an `AcDbEntity`, this setting tells `wblockClone()` to not append the entity. However, as indicated in the previous section, if we are not doing a WBLOCK* and the cloned entity is to occupy model space or paper space, then the default behavior must be overridden and the append must be called.

If you allow the default behavior to occur in the call to `wblockClone()` the entity, its clone will end up in the database, but it will be ownerless. It will not have been appended to its new owner, and there is no current way to complete its append via the API. For the referenced entity to be appended, the `isPrimary` value must be reset to `Adesk::kTrue` before calling its `wblockClone()` function.

The following two cases show how one might handle a hard reference from a custom object to another entity. The first case is simpler, but it requires that the referring and referenced entities always exist in the same `AcDbBlockTableRecord`. The second shows what must be considered if the two entities can exist in different records, or when the reference is in an `AcDbObject` instead of an `AcDbEntity`.

Handling Hard References to AcDbEntities During wblockClone(): CASE 1

If the referring and referenced entities always exist in the same `AcDbBlockTableRecord`, it is sufficient for the referring entity's overridden `wblock()` to forward the `isPrimary` value it received to call the referenced entity's `wblock()`. This takes advantage of the default behavior of all three forms of WBLOCK, as noted in the previous section. We do not need to be concerned with which type of WBLOCK is occurring.

Here are two ways to override the default behavior of wblock clone for this case. First, you could overwrite the entire `wblockClone()` for the referring entity. In the sample code for the default implementation of `wblockClone()` (in the previous section), you'll see a loop on `getNextHardObject()`. Within this loop you would have to intercept the referenced object's ID and change the `isPrimary` value from `Adesk::kFalse` to be the same as the `isPrimary` value passed in.

However, a much simpler way to do this is to continue to use the default `wblockClone()` for your custom entity, but clone the referenced entity first, with the correct settings, when the default setting of `isPrimary` would not be correct. Once you've cloned the referenced entity, when you call your own `wblockClone()`, it will see that the referenced entity is already cloned and will not attempt to clone it using the default settings. The following sample demonstrates this. The data member, `mRefEnt`, is the reference `AcDbHardPointerId`.

```
Acad::ErrorStatus
AsdkEntity::wblockClone(AcRxObject* pOwner,
                        AcDbObject*& pClone,
                        AcDbIdMapping& idMap,
                        Adesk::Boolean isPrimary) const
{
    // If isPrimary is kTrue, then override the default cloning
    // within our own cloning, which would set it to kFalse,
    // by cloning our referenced entity first.
    //
    if (isPrimary) {
        Acad::ErrorStatus es;
        AcDbEntity* pEnt;
        es = acdbOpenAcDbEntity(pEnt, mRefEnt, AcDb::kForRead);

        if (es != Acad::eOk)
            return es;

        // Use the same owner, and pass in the same isPrimary
        // value.
        //
```

```
        AcDbObject* pSubClone = NULL;
        es = pEnt->wblockClone(pOwner, pSubClone, idMap, kTrue);

        if (pSubClone != NULL)
            pSubClone->close();

        pEnt->close();

        if (es != Acad::eOk)
           return es;
    }

    // Now we can clone ourselves by calling our parent's method.
    //
    return AcDbEntity::wblockClone(pOwner, pClone, idMap,
        isPrimary);
}
```

Handling Hard References to AcDbEntities During wblockClone(): CASE 2

The previous example will only work when the reference is in an entity, and that entity is always in the same block table record as the referenced entity. Because they are in the same block table record, the setting of `isPrimary` for the referring entity will also be valid for the referenced entity. However, if the referenced entity can exist in a different block table record, or if the referring entity is an `AcDbObject`, you must use other means to determine if appending should be done.

First, you will need to check the WBLOCK notification to determine which type of WBLOCK is occurring, probably by setting a global flag that can then be queried by your `wblockClone()` function:

- If it is a WBLOCK*, do not use `AcDbBlockTableRecord::appendAcDbEntity()` in the custom class's override of `wblockClone()`, during callbacks, or in any other place.
- If it is a WBLOCK of a user-defined block, it may depend on where the referenced entity currently exists. First, remember that the selected block is getting exploded into model space of the destination drawing. You may want to define this behavior in some other way, but a couple scenarios may be as follows: 1) always clone the referenced entities into model space as well. In this case, you would always set `isPrimary` to `Adesk::kTrue`, or, 2) check the current location of the referenced entity. If it is in model space or paper space, clone it to the corresponding space and set `isPrimary` to `Adesk::kTrue`. If it is in the selected block, also clone it to model space. If it is in some other user-defined block, then call `wblockClone()` on that block record. Just be sure that you do not try to clone the selected block. In this case, the block table record will take care of cloning your referenced entity.

- If it is a WBLOCK of a selection set, only reset `isPrimary` to `Adesk::kTrue` if the referenced entity is going into model space or paper space. If it is in a user-defined block, call `wblockClone()` on that `AcDbBlockTableRecord`, instead of on your referenced entity.

Finally, it should be noted that setting up a hard reference to an `AcDbEntity` is not currently supported by the `AcDbProxyObject` system, even if you use an `AcDbHardPointerId` for the reference. `AcDbProxyObject` uses the default `wblockClone()` implementation, and thus will not do the append of any referenced entities during either form of WBLOCK. If a WBLOCK happens when your entities are proxies, the references will get cloned, but without the append they will be ownerless and are not persistent. The result is that when the wblocked drawing gets loaded, your reference ID will be `NULL`, and the referenced entity will be missing. You must code your custom object to handle this situation gracefully.

Insert

The insert operation is a special case of deep cloning. In the case of an insert, the objects are not copied into the destination database; instead, they are moved into the new database. When this occurs, the source database is no longer valid, because it has been cannibalized when its objects were moved into the new database. If you override the `deepClone()` function, your objects will simply be cloned when an insert operation is called for. If you use the default form of `deepClone()`, cheap cloning is performed internally.

When an object is copied in this way, the ID map still contains two object IDs for each cloned object (the source ID and the destination ID), but these IDs point temporarily to the same object. When the insert operation finishes, the source database is deleted.

Editor Reactor Notification Functions

The `AcEditorReactor` class provides four notification functions that return control to the application at certain points in the deep clone operation. The following functions are called during all deep clone and wblock clone operations:

- `beginDeepClone()`
- `beginDeepCloneXlation()`
- `abortDeepClone()`
- `endDeepClone()`

The `beginDeepClone()` function is called after the `AcDbIdMapping` instance is created and before any objects are cloned. The ID map will be empty, but it can be queried for `destDb()` and `deepCloneContext()` at this time.

The `beginDeepCloneXlation()` function is called after all of the objects in the primary selection set have been cloned and before the references are translated. This is the first time it is possible to see the entire set of what was cloned in the ID map. It is also the time to clone any additional objects and add them to the ID map. Remember that any objects cloned have their object IDs in flux at this point.

The `abortDeepClone()` function is called at any time between `beginDeepClone()` and `endDeepClone()`.

The `endDeepClone()` function is called at the end of the cloning and translation process. The object IDs are no longer in flux. However, this call does not mean that the entities are in their final state for whatever command is being executed. Often the cloned entities are transformed or other operations are performed following the cloning process. There are additional callback functions that can be used to access the entities later, including `commandEnded()`.

In addition to the previous four functions, the following notification functions are provided in the wblock clone operation:

- `beginWblock()`
- `otherWblock()`
- `abortWblock()`
- `endWblock()`

These calls come in the following order with the deep clone functions:

1. `beginDeepClone()` This call is sent as soon as the destination `AcDbDatabase` instance has been created, but it is in a "raw" state and is not ready for appending.
2. `beginWblock()` The new database now has its basic elements, such as a handle table, a class ID map, and model space and paper space block table records. It is still empty. The cloning has not begun, but the new database is now ready for appending.
3. `otherWblock()` and `beginDeepCloneXlation()` These two calls are made back-to-back and can be used for the same purpose. The primary set of objects has been cloned, but the reference translation has not started yet.
4. `endDeepClone()` The translation process has now completed, but the entities are not yet in their final state.
5. `endWblock()` The entities have now been transformed, and the model space and paper space origins have been set. The new database is complete but has not yet been saved.

There are three types of `AcEditorReactor::beginWblock()`. They are listed here along with their corresponding `AcDbDatabase` functions:

1. WBLOCK*

```
void
AcEditorReactor::beginWblock(
    AcDbDatabase* pTo,
    AcDbDatabase* pFrom)

Acad::ErrorStatus
AcDbDatabase::wblock(AcDbDatabase*& pOutputDatabase)
```

2. WBLOCK of a user-defined block

```
void
AcEditorReactor::beginWblock(
    AcDbDatabase* pTo,
    AcDbDatabase* pFrom,
    AcDbObjectId blockId)

Acad::ErrorStatus
AcDbDatabase::wblock(
    AcDbDatabase*& pOutputDatabase,
    AcDbObjectId nObjId)
```

3. WBLOCK of a selection set

```
void
AcEditorReactor::beginWblock(
    AcDbDatabase* pTo,
    AcDbDatabase* pFrom,
    const AcGePoint3d& insertionPoint)

Acad::ErrorStatus
AcDbDatabase::wblock(
    AcDbDatabase*& pOutputDatabase,
    const AcDbObjectIdArray& pIdSet,
    const AcGePoint3d& pPoint3d)
```

All three versions clone both the model space and paper space `AcDbBlockTableRecord` before calling `beginWblock()`. However, for the entities within these block table records, the order of notification will appear to come differently in the first type and last two types. In version one, entities in model space that are being cloned will receive the call for `wblockClone()` before the `AcEditorReactor::beginWblock()`. In versions two and three, entities in the `AcDbBlockTableRecord` or the selection set will get their `wblockClone()` call after the `AcEditorReactor::beginWblock()` notification call.

Objects that have been cloned during a partial XBIND are automatically redirected just after `endDeepClone()` notification. This means that their `AcDbObjectIds` in the externally referenced database are forwarded to the `AcDbObjectIds` of the clone objects in the host drawing, and the objects in

the externally referenced database are deleted. Objects that reference the forwarded `AcDbObjectIds` end up referencing the clones in the host drawing. If you need to disable this automatic redirection for your objects, then remove the `idPair()` from the `idMap`, for your cloned objects, during `endDeepClone()` notification.

The following function calls occur during an INSERT or INSERT* command:

- `beginInsert()`
- `otherInsert()`
- `abortInsert()`
- `endInsert()`

These calls come in the following order with the deep clone functions:

1. `beginInsert()` and `beginDeepClone()` These calls come back-to-back and can be used for the same purpose.
2. `otherInsert()` and `beginDeepCloneXlation()` These calls also come back-to-back and can be used for the same purpose.
3. `endDeepClone()` The cloning and translation processes are completed. The entities are cloned but have not been appended to a block, so they are not graphical. You cannot use the entities in a selection set yet.
4. `endInsert()` The entities have now been transformed and have been appended to a block. If this is an INSERT*, they are now in model space and have their graphics. They can be used in selection sets. However, if this is an INSERT, they have only been appended to a block table record; that record has not yet been added to the block table. In this case, you must wait until `commandEnded()` notification to use these entities in a selection set.

The sample code in this section uses the `beginDeepCloneXlation()` notification function. This sample illustrates how you could write a reactor to add behavior to the WBLOCK command to tell it to include all text styles in the new drawing, instead of only the text styles that are referenced by the entities. It thus shows how to use wblock with nonentities.

`AcDbIdMapping` has a function, `deepCloneContext()`, which returns the context in which the deep clone function was called. The contexts are the following:

kDcCopy	Copying within a database; uses COPY, ARRAY, MIRROR (if you are not deleting the original), LEADER acquisition, or copy of an INSERT
kDcExplode	EXPLODE of a block reference
kDcBlock	BLOCK creation
kDcXrefBind	XREF Bind and XBIND
kDcSymTable Merge	XREF Attach, DXFIN, and IGESIN (only symbol table records are cloned here)
kDcSaveAs	SAVEAS when VISRETAIN is set to 1 (only symbol table records are cloned here)
kDcInsert	INSERT of a drawing
kDcWblock	WBLOCK
kDcObjects	`AcDbDatabase::deepCloneObjects()`

The `AcEditorReactor::abortDeepClone()` function is called when a call to `AcDbDatabase::abortDeepClone()` is made.

The following code uses a transient editor reactor derived from `AcEditorReactor` and overrides the `beginDeepCloneXlation()` function for the reactor.

```
//  Since AcDbDatabase::wblock() only supports AcDbEntities
//  in its array of IDs, this code demonstrates how to add
//  additional objects during beginDeepCloneXlation(). If
//  it is a WBLOCK command, it asks the user if all text
//  styles should be wblocked.  Otherwise, only those text
//  styles referenced by entities being wblocked
//  will be included (wblock's default behavior).

// AsdkEdReactor is derived from AcEditorReactor.
//
void
AsdkEdReactor::beginDeepCloneXlation(AcDbIdMapping& idMap,
    Acad::ErrorStatus* es)
{
    if (idMap.deepCloneContext() == AcDb::kDcWblock
        && getYorN("Wblock all Text Styles"))
    {
        AcDbDatabase *pOrigDb, *pDestDb;
        if (idMap.origDb(pOrigDb) != Acad::eOk)
            return;
```

```
*es = idMap.destDb(pDestDb);
if (*es != Acad::eOk)
    return;

AcDbTextStyleTable *pTsTable;
*es = pOrigDb->getSymbolTable(pTsTable,
    AcDb::kForRead);
if (*es != Acad::eOk)
    return;

AcDbTextStyleTableIterator *pTsIter;
*es = pTsTable->newIterator(pTsIter);

if (*es != Acad::eOk) {
    pTsTable->close();
    return;
}

AcDbTextStyleTableRecord *pTsRecord;
AcDbObject *pClonedObj;

for (; !pTsIter->done(); pTsIter->step()) {
    *es = pTsIter->getRecord(pTsRecord,
        AcDb::kForRead);
    if (*es != Acad::eOk) {
        delete pTsIter;
        pTsTable->close();
        return;
    }

    // It is not necessary to check for already cloned
    // records.  If the text style is already
    // cloned, wblockClone() will return Acad::eOk
    // and pCloneObj will be NULL.
    //
    pClonedObj = NULL;
    *es = pTsRecord->wblockClone(pDestDb,
        pClonedObj, idMap, Adesk::kFalse);

    if (*es != Acad::eOk) {
        pTsRecord->close();
        delete pTsIter;
        pTsTable->close();
        return;
    }

    *es = pTsRecord->close();

    if (*es != Acad::eOk) {
        delete pTsIter;
        pTsTable->close();
        return;
    }
```

```
            if (pClonedObj != NULL) {
                *es = pClonedObj->close();
                if (*es != Acad::eOk) {
                    delete pTsIter;
                    pTsTable->close();
                    return;
                }
            }
        }
        delete pTsIter;
        *es = pTsTable->close();
    }
}
```

Protocol Extension

19

In This Chapter

- Protocol Extension Defined
- Implementing Protocol Extension
- Protocol Extension for the MATCH Command
- Protocol Extension Example

All C++ class definitions are fixed at compile time. Under normal circumstances, if you write a file translator or an editing command that operates on a number of existing AutoCAD classes, you have to redefine all the existing classes to include the new translator or editing functions. And you would have to recompile your library as well as all the applications that use it.

By using the ObjectARX protocol extension mechanism described in this chapter, you can add functionality to existing ObjectARX classes at runtime, without any modification of existing classes and recompilation.

Protocol Extension Defined

Protocol extension is a mechanism for adding functionality to existing ObjectARX classes. This new functionality is embodied in protocol extension classes associated with an existing ObjectARX class at runtime. The association of an ObjectARX class with protocol extension classes is made by way of the ObjectARX class descriptor object (described in chapter 11, "Deriving a Custom ObjectARX Class.") The class descriptor object describes the class and includes an array of pointers to any objects that extend the functionality of that class. An ObjectARX class can have any number of protocol extensions associated with it.

Implementing Protocol Extension

To implement protocol extension

1. Declare and define the protocol extension classes that include the additional functionality.
2. Register the protocol extension classes with the application, and associate them with the ObjectARX classes to which they add functionality.

These steps are described in detail in the next two subsections. Additional information about implementation issues follows these sections.

Declaring and Defining Protocol Extension Classes

The example included at the end of this chapter provides a simple illustration of protocol extension. It defines a "temperature" property protocol extension class. A default implementation is defined for `AcDbEntity`, and specific implementations are defined for `AcDbCircle` and `AcDbRegion`.

The example serves as a model for more complex (and realistic) uses of the ObjectARX protocol extension mechanism. The base class for this protocol extension, called `AsdkEntTemperature`, is derived from `AcRxObject`. This class defines the virtual functions that will be inherited by the derived protocol extension classes, `AsdkDefaultTemperature`, `AsdkCircleTemperature`,

and `AsdkRegionTemperature`. In this example, there is only one function: `reflectedEnergy()`. The class hierarchy for the protocol extension classes is shown in the following figure:

The first step in using protocol extension is to declare and define each of the protocol extension classes. The base class, `AsdkEntTemperature`, is an abstract base class that is defined using the `ACRX_NO_CONS_DEFINE_MEMBERS()` macro. This class will eventually be registered as part of the ObjectARX class hierarchy.

The child classes are defined using standard C++ syntax for deriving new classes. These classes should not be registered in the ObjectARX class hierarchy, so you don't need to use the ObjectARX macros for them.

For each class, you implement the functions that constitute the protocol extension. In this example, each class has only one function, `reflectedEnergy()`, which calculates a temperature for the entity.

Registering Protocol Extension Classes

To register protocol extension classes with your application

1 Initialize your new protocol extension parent class and add it to the runtime class hierarchy as shown in the following example:

```
AsdkEntTemperature::rxInit();
acrxBuildClassHierarchy();
```

These function calls are required for any new ObjectARX class, as described in chapter 11, "Deriving a Custom ObjectARX Class."

2 Create an object of each protocol extension class and add the objects to the appropriate `AcRxClass` descriptor objects using the `addX()` function as shown in the following example:

```
pDefaultTemp = new AsdkDefaultTemperature();
pRegionTemp = new AsdkRegionTemperature();
pCircleTemp = new AsdkCircleTemperature();

// Add the protocol extension objects to the appropriate
// AcRxClass objects.
//
AcDbEntity::desc()->addX(AsdkEntTemperature::desc(),
    pDefaultTemp);
AcDbRegion::desc()->addX(AsdkEntTemperature::desc(),
    pRegionTemp);
AcDbCircle::desc()->addX(AsdkEntTemperature::desc(),
    pCircleTemp);
```

At runtime, ObjectARX constructs a class descriptor object structure that includes the basic ObjectARX class hierarchy as well as the protocol extension objects associated with the ObjectARX class descriptor objects. The following figure shows the class descriptor object structure for the classes that relate to the `AsdkEntTemperature` example in this chapter:

Default Class for Protocol Extension

It is recommended that you implement a default protocol extension class, as shown in the example at the end of this chapter. If there is no corresponding protocol extension object for a particular class, ObjectARX searches up the class hierarchy and uses the nearest one that it finds. It is also recommended that you associate the default protocol extension class with `AcRxObject` or with some other class at the top of the ObjectARX class hierarchy such as `AcDbEntity` (in this example) or `AcDbObject`.

Unloading the Application

When the application is unloaded, you need to remove any commands that were added at initialization. In addition, you should remove your protocol extension class from the ObjectARX class dictionary and delete the class descriptor object for your protocol extension class.

Using Protocol Extension Functionality in an Application

To make use of protocol extension functionality, you need to obtain the class descriptor object for a particular class. Once you have obtained a pointer to the class descriptor object, you can call any of the methods for that class. The following is an example of using the `AsdkEntTemperature` protocol extension for the `AcDbEntity` class:

```
AcDbEntity *pEnt;
AsdkEntTemperature *pTemp;

pTemp = AsdkEntTemperature::cast(
    pEnt->x(AsdkEntTemperature::desc()));
double eTemp = pTemp -> reflectedEnergy (pEnt);
```

You can use the `ACRX_X_CALL` macro to simplify this code as follows:

```
double eTemp = ACRX_X_CALL(pEnt,
     AsdkEntTemperature)->reflectedEnergy(pEnt);
```

Protocol Extension for the MATCH Command

`AcDbMatchProperties` is an ObjectARX class specifically designed to serve as a protocol extension base class that provides entity support for the MATCHPROP command. (See the AutoCAD *Command Reference* for information about MATCHPROP.) The default protocol extension class for `AcDbEntity` allows copying of color, layer, linetype, and linetype scale properties from one entity to another. It is, nevertheless, preferable to implement `AcDbMatchProperties` as a protocol extension class for all custom objects derived from `AcDbEntity`, to provide full support for MATCHPROP. If the default protocol extension class is overridden with `AcDbMatchProperties`, it must include functions to copy the base class properties as well.

Protocol Extension Example

This protocol extension example is divided into three parts:

- Declaration and definition of four protocol extension classes: `AsdkEntTemperature`, `AsdkDefaultTemperature`, `AsdkRegionTemperature`, and `AsdkCircleTemperature`.
- The implementation of the `energy()` function for the ENERGY command, which allows the user to select an entity and then calculates a temperature for that entity.
- The ObjectARX module interface functions: `initApp()`, `unloadApp()`, and `acrxEntryPoint()`.

```
// This is the AsdkEntTemperature protocol extension abstract base
// class. Notice that this is the lowest level that uses
// the ACRX macros.
//
class AsdkEntTemperature : public AcRxObject
{
public:
    ACRX_DECLARE_MEMBERS(AsdkEntTemperature);
    virtual double reflectedEnergy(AcDbEntity*) const = 0;
};

ACRX_NO_CONS_DEFINE_MEMBERS(AsdkEntTemperature, AcRxObject);
```

```
// This is the default implementation to be attached to AcDbEntity
// as a catch-all. This guarantees that this protocol extension will
// be found for any entity, so the search up the AcRxClass tree will
// not fail and abort AutoCAD.
//
class AsdkDefaultTemperature : public AsdkEntTemperature

{
public:
    virtual double reflectedEnergy(AcDbEntity* pEnt) const;
};

double
AsdkDefaultTemperature::reflectedEnergy(
    AcDbEntity* pEnt) const
{
    acutPrintf(
        "\nThis entity has no area, and no reflection.\n");
    return -1.0;
}

// AsdkEntTemperature implementation for Regions
//
class AsdkRegionTemperature : public AsdkEntTemperature
{
public:
    virtual double reflectedEnergy(AcDbEntity* pEnt) const;
};

double
AsdkRegionTemperature::reflectedEnergy(
    AcDbEntity* pEnt) const
{
    AcDbRegion *pRegion = AcDbRegion::cast(pEnt);
    if (pRegion == NULL)
        acutPrintf("\nThe impossible has happened!");

    // Compute the reflected energy as the region area multiplied
    // by a dummy constant.
    //
    double retVal;
    if (pRegion->getArea(retVal) != Acad::eOk)
        return -1.0;
    return retVal * 42.0;
}

// AsdkEntTemperature implementation for circles
//
class AsdkCircleTemperature : public AsdkEntTemperature
{
public:
    virtual double reflectedEnergy(AcDbEntity* pEnt) const;
};
```

```
double
AsdkCircleTemperature::reflectedEnergy(
    AcDbEntity* pEnt) const
{
    AcDbCircle *pCircle = AcDbCircle::cast(pEnt);
    // Compute the reflected energy in a manner distinctly
    // different than for AcDbRegion.
    //
    return pCircle->radius() * 6.21 * 42.0;
}

// This function has the user select an entity and then
// calls the reflectedEnergy() function in the protocol
// extension class attached to that entity's class.
//
void
energy()
{
    AcDbEntity *pEnt;
    AcDbObjectId pEntId;
    ads_name en;
    ads_point pt;
    if (acedEntSel("\nSelect an Entity:  ", en, pt)
        != RTNORM)
    {
        acutPrintf("Nothing Selected\n");
        return;
    }

    acdbGetObjectId(pEntId, en);
    acdbOpenObject(pEnt, pEntId, AcDb::kForRead);

    // call the protocol extension class's method
    //
    double eTemp = ACRX_X_CALL(pEnt,
        AsdkEntTemperature)->reflectedEnergy(pEnt);
    acutPrintf("\nEnergy == %f\n", eTemp);
    pEnt->close();
}

// Pointers for protocol extension objects. These pointers
// are global so that they can be accessed during
// initialization and cleanup.
//
AsdkDefaultTemperature *pDefaultTemp;
AsdkRegionTemperature *pRegionTemp;
AsdkCircleTemperature *pCircleTemp;

// Initialization function called from acrxEntryPoint() during
// kInitAppMsg case. This function is used to add commands
// to the command stack and to add protocol extension
// objects to classes.
//
```

```
void
initApp()
{
    acrxRegisterService("AsdkTemperature");
    AsdkEntTemperature::rxInit();
    acrxBuildClassHierarchy();

    pDefaultTemp = new AsdkDefaultTemperature();
    pRegionTemp = new AsdkRegionTemperature();
    pCircleTemp = new AsdkCircleTemperature();

    // Add the protocol extension objects to the appropriate
    // AcRxClass objects.
    //
    AcDbEntity::desc()->addX(AsdkEntTemperature::desc(),
        pDefaultTemp);
    AcDbRegion::desc()->addX(AsdkEntTemperature::desc(),
        pRegionTemp);
    AcDbCircle::desc()->addX(AsdkEntTemperature::desc(),
        pCircleTemp);
    acedRegCmds->addCommand("ASDK_TEMPERATURE_APP",
        "ASDK_ENERGY", "ENERGY", ACRX_CMD_TRANSPARENT,
        energy);
}

void
unloadApp()
{
    delete acrxServiceDictionary->remove("AsdkTemperature");
    acedRegCmds->removeGroup("ASDK_TEMPERATURE_APP");

    // Remove protocol extension objects from the AcRxClass
    // object tree.  This must be done before removing the
    // AsdkEntTemperature class from the ACRX runtime class
    // hierarchy, so the AsdkEntTemperature::desc()
    // still exists.
    //
    AcDbEntity::desc()->delX(AsdkEntTemperature::desc());
    delete pDefaultTemp;
    AcDbRegion::desc()->delX(AsdkEntTemperature::desc());
    delete pRegionTemp;
    AcDbCircle::desc()->delX(AsdkEntTemperature::desc());
    delete pCircleTemp;

    // Remove the AsdkEntTemperature class from the ARX
    // runtime class hierarchy.
    //
    deleteAcRxClass(AsdkEntTemperature::desc());
}
```

```
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
    }
    return AcRx::kRetOK;
}
```

ObjectARX Global Utility Functions

In This Chapter

20

- Common Characteristics of ObjectARX Library Functions
- Variables, Types, and Values Defined in ObjectARX
- Lists and Other Dynamically Allocated Data
- Extended Data Exclusive Data Types
- Text String Globalization Issues

This chapter discusses some general characteristics of the ObjectARX global utility functions. For more information on specific functions, see the *ObjectARX Reference*.

Common Characteristics of ObjectARX Library Functions

This section describes some general characteristics of global functions in the ObjectARX library. Most ObjectARX global functions that operate on the database, system variables, and selection sets work on the current document.

NOTE The functions described in this chapter were known as the ADS functions in previous releases of AutoCAD.

ObjectARX Global Function Calls Compared to AutoLISP Calls

Many ObjectARX global functions are unique to the ObjectARX programming environment, but many provide the same functionality as AutoLISP functions: they have the same name as the comparable AutoLISP function, except for the prefix (aced, acut, etc.). This similarity makes it easy to convert programs from AutoLISP to ObjectARX. However, there are important differences between the interpretive AutoLISP environment and the compiled C++ environment.

Argument Lists in AutoLISP and C

Many built-in AutoLISP functions accept an arbitrary number of arguments. This is natural for the LISP environment, but to require variable-length argument lists for every comparable function in the ObjectARX library would impose needless complexity. To avoid this problem, a simple rule was applied to the library: an ObjectARX function that is an analog of an AutoLISP function takes all arguments that the AutoLISP function takes. Where an argument is optional in AutoLISP, in the ObjectARX library a special value, usually a null pointer, 0, or 1, can be passed to indicate that the option is not wanted.

A few ObjectARX library functions are exceptions to this rule. The `acutPrintf()` function is similar to the standard C library `printf()` function. Like the standard version, it is implemented as a variadic function; that is, it takes a variable-length argument list. The `acedCommand()` and `acedCmd()` functions are more complex. The AutoLISP `command` function not only accepts a variable number of arguments of various types, but it also accepts types defined especially for AutoCAD, such as points and selection sets. To achieve the same flexibility in ObjectARX, `acedCommand()` takes a

variable-length argument list and in addition takes arguments to specify the type of the values being passed; `acedCmd()` requires a similar set of values but is passed as a linked list. Therefore, `acedCommand()` and `acedCmd()` arguments do not correspond exactly to the AutoLISP `command` function. Finally, the AutoLISP `entget` function has an optional argument for retrieving extended data. In ObjectARX, the `acdbEntGet()` function does not have a corresponding argument. Instead, there is an additional function, `acdbEntGetX()`, provided specifically for retrieving extended data.

Memory Considerations

The memory requirements of an ObjectARX application are different from those of AutoLISP. On the one hand, the data structures employed by C++ programs tend to be more compact than AutoLISP lists. On the other hand, there is a rather large, fixed overhead for running ObjectARX applications. Part of this consists of code that must be present in the applications themselves; the larger part is the ObjectARX library.

Memory Management

Some ObjectARX global functions allocate memory automatically. In most cases, the application must explicitly release this memory as if the application itself had allocated it. AutoLISP has automatic garbage collection, but ObjectARX does not.

WARNING! Failure to do this slows down the system and can cause AutoCAD to terminate.

Function Return Values versus Function Results

A few ObjectARX global functions have a `void` return type, and some directly return their results, but most are of `int` type and return an integer status code that indicates whether the function call succeeded or failed.

The code `RTNORM` indicates that the function succeeded; other codes indicate failure or special conditions. Library functions that return a status code pass their actual results (if any) back to the caller through an argument that is passed by reference.

NOTE Do not confuse a library function's result arguments and values with its return value. The function returns an integer status code. It places its results in arguments passed (by reference) back to the function that calls it.

Consider the following prototyped declarations for a few typical ObjectARX functions:

```
int acdbEntNext(ads_name ent, ads_name result);

int acedOsnap(ads_point pt, char *mode, ads_point
    result);

int acedGetInt(char *prompt, int *result);
```

An application could call these functions with the following C++ statements:

```
stat = acdbEntNext(ent, entres);

stat = acedOsnap(pt, mode, ptres);

stat = acedGetInt(prompt, &intres);
```

After each function is called, the value of the stat variable indicates either success (`stat == RTNORM`) or failure (`stat == RTERROR` or another error code, such as `RTCAN` for cancel). The last argument in each list is the result argument, which must be passed by reference. If successful, `acdbEntNext()` returns an entity name in its `entres` argument, `acedOsnap()` returns a point in `ptres`, and `acedGetInt()` returns an integer result in `intres`. (The types `ads_name` and `ads_point` are array types, which is why the `entres` and `ptres` arguments don't explicitly appear as pointers.)

NOTE In ObjectARX global function declarations, the result arguments always follow the arguments that pass input values to the function.

External Functions

Once an ObjectARX application has defined its external functions (with calls to `acedDefun()`), the functions can be called by the AutoLISP user and by AutoLISP programs and functions as if they were built-in or user-defined AutoLISP functions. An external function can be passed AutoLISP values and variables, and can return a value to the AutoLISP expression that calls it. Some restrictions apply and are described in this section.

Defining External Functions

When an ObjectARX application receives a `kLoadDwgMsg` request from AutoCAD, it must define all of its external functions by calling `acedDefun()` once for each function. The `acedDefun()` call associates the external function's name (passed as a string value) with an integer code that is unique within the application. The integer code must not be negative, and it cannot be greater than 32,767 (in other words, the code is a short integer).

The following call to `acedDefun()` specifies that AutoLISP will recognize an external function called `doit` in AutoLISP, and that when AutoLISP invokes `doit`, it passes the function code zero (0) to the ObjectARX application:

```
acedDefun("doit", 0);
```

The string that specifies the name of the new external function can be any valid AutoLISP symbol name. AutoLISP converts it to all uppercase and saves it as a symbol of the type `Exsubr`.

External functions are defined separately for each open document in the MDI. The function gets defined when the document becomes active. For more information, see chapter 16, "The Multiple Document Interface."

WARNING! If two or more ObjectARX applications define functions (in the same document) that have the same name, AutoLISP recognizes only the most recently defined external function. The previously loaded function will be lost. This can also happen if the user calls `defun` with a conflicting name.

As in AutoLISP, the new function can be defined as an AutoCAD command by prefixing its name with "C:" or "c:", as shown in the following example:

```
acedDefun("C:DOIT", 0);
```

In this case, DOIT can now be invoked from the AutoCAD Command prompt without enclosing its name in parentheses.

Functions defined as AutoCAD commands can still be called from AutoLISP expressions, provided that the "C:" prefix is included as a part of their names. For example, given the previous `acedDefun()` call, the AutoCAD user could also invoke the DOIT command as a function with arguments:

Command: **(c:doit x y)**

WARNING! If the application defines a `c:xxx` command whose name conflicts with a built-in command or a command name defined in the *acad.pgp* file, AutoCAD does not recognize the external function as a command. The function can still be invoked as an AutoLISP external function. For example, after the call `acedDefun("c:cp", 0)`, a user input of `cp` (an alias for COPY defined in *acad.pgp*) invokes the AutoCAD COPY command, but the user could invoke the external function with `c:cp`.

NOTE Function names defined by `acedDefun()` can be undefined by calling `acedUndef()`. After a function has been undefined, an attempt to invoke it causes an error.

Evaluating External Functions

Once an external function has been defined, AutoLISP can invoke it with an `kInvkSubrMsg` request. When the ObjectARX application receives this request, it retrieves the external function's integer code by calling `acedGetFunCode()`. Then a switch statement, an if statement, or a function-call table can select and call the indicated function handler. This is the function that the ObjectARX application defines to implement the external function. Note that the name of the handler and the name defined by `acedDefun()` (and therefore recognized by AutoLISP) are not necessarily the same name.

If the function handler expects arguments, it can retrieve their values by calling `acedGetArgs()`, which returns a pointer to a linked list of result buffers that contain the values passed from AutoLISP. If the handler expects no arguments, it does not need to call `acedGetArgs()` (it can do so anyway, to verify that no arguments were passed). Because it retrieves its arguments from a linked list, the function handler can also implement variable-length argument lists or varying argument types.

NOTE The function handler must verify the number and type of arguments passed to it, because there is no way to tell AutoLISP what the requirements are.

Function handlers that expect arguments can be written so that they prompt the user for values if `acedGetArgs()` returns a `NULL` argument list. This technique is often applied to external functions defined as AutoCAD commands.

A group of ObjectARX functions known as value-return functions (such as `acedRetInt()`, `acedRetReal()`, and `acedRetPoint()`) enable an external function to return a value to the AutoLISP expression that invoked it.

Arguments that are passed between external functions and AutoLISP must evaluate to one of the following types: integer, real (floating-point), string, point (represented in AutoLISP as a list of two or three real values), an entity name, a selection set name, the AutoLISP symbols `t` and `nil`, or a list that contains the previous elements. AutoLISP symbols other than `t` and `nil` are not passed to or from external functions, but an ObjectARX application can retrieve and set the value of AutoLISP symbols by calling `acedGetSym()` and `acedPutSym()`.

If, for example, an external function in an ObjectARX application is called with a string, an integer, and a real argument, the AutoLISP version of such a function can be represented as follows:

```
(doitagain pstr iarg rarg)
```

Assuming that the function has been defined with `acedDefun()`, an AutoCAD user can invoke it with the following expression:

Command: **(doitagain "Starting width is" 3 7.12)**

This call supplies values for the function's string, integer, and real number arguments, which the `doitagain()` function handler retrieves by a call to `acedGetArgs()`. For an example of retrieving arguments in this way, see the first example in "Lists and Other Dynamically Allocated Data" on page 546.

Error Handling

The AutoCAD environment is complex and interactive, so ObjectARX applications must be robust. ObjectARX provides several error-handling facilities. The result codes returned during "handshaking" with AutoLISP indicate error conditions, as do the result codes library functions returned to the application. Functions that prompt for input from the AutoCAD user employ the built-in input-checking capabilities of AutoCAD. In addition, three functions let an application notify users of an error: `acdbFail()`, `acedAlert()`, and `acrxAbort()`.

The `acdbFail()` function simply displays an error message (passed as a single string) at the AutoCAD Command prompt. This function can be called to identify recoverable errors such as incorrect argument values passed by the user.

The statement in the following example calls `acdbFail()` from a program named *test.arx*:

```
acdbFail("invalid osnap point\n");
```

The `acdbFail()` function displays the following:

Application test.arx ERROR: invalid osnap point

You can also warn the user about error conditions by displaying an alert box. To display an alert box, call `acedAlert()`. Alert boxes are a more emphatic way of warning the user, because the user has to choose OK before continuing.

For fatal errors, `acrxAbort()` should be called. This function prompts the user to save work in progress before exiting. The standard C++ `exit()` function should not be called.

To obtain detailed information about the failure of an ObjectARX function, inspect the AutoCAD system variable ERRNO. When certain ObjectARX function calls (or AutoLISP function calls) cause an error, ERRNO is set to a value that the application can retrieve by a call to `acedGetVar()`. ObjectARX

defines symbolic names for the error codes in the header file *ol_errno.h*, which can be included by ObjectARX applications that examine ERRNO. These codes are shown in the *ObjectARX Reference*. The sample ObjectARX application, `ads_perr`, displays error messages based on the value of ERRNO.

Communication between Applications

The ObjectARX function `acedInvoke()` in one application is used to call external functions defined and implemented by other ObjectARX applications. The external function called by `acedInvoke()` must be defined by a currently loaded ObjectARX application.

The `acedInvoke()` function calls the external function by the name that its application has specified in the `acedDefun()` call, which is the function name that AutoLISP calls to invoke the function. If the external function was defined as an AutoLISP command, with "C:" as a prefix to its name, these characters must be included in the string that `acedInvoke()` specifies (as when the command is invoked with an AutoLISP expression).

WARNING! Because applications loaded at the same time cannot have duplicate function names, you should take this into account when designing an application that uses more than a single program file; avoid the problem with a naming scheme or convention that ensures that the name of each external function will be unique. The best solution is to use your Registered Developer Symbol (RDS) as a prefix.

The name of the external function, and any argument values that it requires, is passed to `acedInvoke()` in the form of a linked list of result buffers. It also returns its result in a result-buffer list; the second argument to `acedInvoke()` is the address of a result-buffer pointer.

The following sample function calls `acedInvoke()` to invoke the factorial function `fact()` in the sample program *fact.cpp*:

```
static void test()
{
    int stat, x = 10;
    struct resbuf *result = NULL, *list;

    // Get the factorial of x from file fact.cpp.
    list = acutBuildList(RTSTR, "fact", RTSHORT, x, RTNONE);

    if (list != NULL) {
        stat = acedInvoke(list, &result);
        acutRelRb(list);
    }
```

```
    if (result != NULL) {
        acutPrintf("\nSuccess: factorial of %d is %d\n", x,
            result->resval.rint);
        acutRelRb(result);
    }
    else
        acutPrintf("Test failed\n");
}
```

If a function is meant to be called with `acedInvoke()`, the application that defines it should register the function by calling `acedRegFunc()`. (In some cases the `acedRegFunc()` call is required, as described later in this section.) When `acedRegFunc()` is called to register the function, ObjectARX calls the function directly, without going through the application's dispatch loop. To define the function, call `acedRegFunc()`.

An external function handler registered by `acedRegFunc()` must have no arguments and must return an integer (which is one of the application result codes—either `RSRSLT` or `RSERR`).

The following excerpt shows how the `funcload()` function in *fact.cpp* can be modified to register its functions as well as define them:

```
typedef int (*ADSFUNC) (void);

// First, define the structure of the table: a string
// giving the AutoCAD name of the function, and a pointer to
// a function returning type int.

struct func_entry { char *func_name; ADSFUNC func; };

// Declare the functions that handle the calls.
int fact (void); // Remove the arguments
int squareroot (void);

// Here we define the array of function names and handlers.
//
static struct func_entry func_table[] =
    {  {"fact", fact},
    {"sqr", squareroot},
    };
...

static int funcload()
{
    int i;
```

```
    for (i = 0; i < ELEMENTS(func_table); i++) {
        if (!acedDefun(func_table[i].func_name, i))
            return RTERROR;
        if (!acedRegFunc(func_table[i].func, i))
            return RTERROR;
    }
    return RTNORM;
}
```

As the code sample shows, the first argument to `acedRegFunc()` is the function pointer (named after the function handler defined in the source code), and not the external function name defined by `acedDefun()` and called by AutoLISP or `acedInvoke()`. Both `acedDefun()` and `acedRegFunc()` pass the same integer function code `i`.

If a registered function is to retrieve arguments, it must do so by making its own call to `acedGetArgs()`.

The `acedGetArgs()` call is moved to be within the function `fact()`. The result-buffer pointer `rb` is made a variable rather than an argument. (This doesn't match the call to `fact()` in the `dofun()` function. If all external functions are registered, as this example assumes, the `dofun()` function can be deleted completely; see the note that follows this example.) The new code is shown in boldface type:

```
static int fact()
{
    int x;
    struct resbuf *rb;

    rb = acedGetArgs();
    if (rb == NULL)
        return RTERROR;

    if (rb->restype == RTSHORT) {
        x = rb->resval.rint; // Save in local variable.
    } else {
        acdbFail("Argument should be an integer.");
        return RTERROR;
    }

    if (x < 0) { // Check the argument range.
        acdbFail("Argument should be positive.");
        return RTERROR;
    } else if (x > 170) { // Avoid floating-point overflow.
        acdbFail("Argument should be 170 or less.");
        return RTERROR;
    }
    acedRetReal(rfact(x)); // Call the function itself, and
                           // return the value to AutoLISP.
    return RTNORM;
}
```

A comparable change would have to be made to `squareroot()`.

NOTE If an application calls `acedRegFunc()` to register a handler for every external function it defines, it can assume that these functions will be invoked by `acedInvoke()`, and it can omit the `kInvkSubrMsg` case in its `acrxEntryPoint()` function. If you design an application that requires more than a single ObjectARX code file, this technique is preferable, because it places the burden of handling function calls on the ObjectARX library rather than on the `acrxEntryPoint()` function.

If a function call starts a calling sequence that causes a function in the same application to be called with `acedInvoke()`, the latter function must be registered by `acedRegFunc()`. If the called function isn't registered, `acedInvoke()` reports an error. The following figure illustrates this situation:

In the illustration above,

- `A_tan()` invokes `B_sin()`
- `A_tan()` invokes `C_cos()`
- `B_sin()` invokes `A_pi()`
- `C_cos()` invokes `A_pi()`

where application A defines `A_tan()` and `A_pi()`, application B defines `B_sin()`, and application C defines `C_cos()`. The `A_pi()` function must be registered by `acedRegFunc()`.

To prevent `acedInvoke()` from reporting registration errors, register any external function that is meant to be called with `acedInvoke()`.

The `acedRegFunc()` function can be called also to unregister an external function. The same application must either register or unregister the function; ObjectARX prohibits an application from directly managing another application.

Handling Errors from Invoked Functions

When `acedInvoke()` returns `RTNORM`, this implies that the external function was called and returned successfully. It does not imply that the external function successfully obtained a result; to obtain this information, your program must inspect the result argument. If the external function is successful and is meant to return values, result points to a result-buffer list containing one or more values. If the external function failed, the result argument is set to `NULL`. The result argument is also `NULL` if the external function doesn't return a result.

The following sample code fragment checks the return value of an external function that is expected to return one or more result values:

```
struct resbuf *xfcnlist, *xresults;

// Build the invocation list, xfcnlist.

rc = acedInvoke(xfcnlist, &xresults);
if (rc != RTNORM) {
// Couldn't call the function—report this error (or even abort).
    return BAD;
}

if (xresults == NULL) {
    // Function was called but returned a bad result.
    return BAD;
}

// Look at return results and process them.
```

Handling External Applications

ObjectARX applications can load and unload other ObjectARX applications and obtain a list of which external applications are currently loaded, just as AutoLISP programs can (using `arxloaded`). The following call loads a program called *myapp*:

```
if (acedArxLoad("myapp") != RTERROR) {
    // Use acedInvoke() to call functions in "myapp".
}
```

When your program is finished with *myapp*, it can unload it by calling `acedArxUnload()`:

```
acedArxUnload("myapp");
```

The function `acedArxLoaded()` can be used to obtain the names of all currently loaded applications, as in the following code:

```
struct resbuf *rb1, *rb2;

for (rb2 = rb1 = acedArxLoaded(); rb2 != NULL; rb2 = rb2->rbnext) {
    if (rb2->restype == RTSTR)
        acutPrintf("%s\n", rb2->resval.rstring);
}
acutRelRb(rb1);
```

You can call the functions `acedArxLoaded()` and `acedArxUnload()` in conjunction with each other. The following example unloads all applications except the current one:

```
struct resbuf *rb1, *rb2;

for (rb2 = rb1 = acedArxLoaded(); rb2 != NULL;
    rb2 = rb2->rbnext) {
    if (strcmp(ads_appname, rb2->resval.rstring) != 0)
        acedArxUnload(rb2->resval.rstring);
}
acutRelRb(rb1);
```

Variables, Types, and Values Defined in ObjectARX

ObjectARX defines a few data types for the AutoCAD environment. It also defines a number of symbolic codes for values passed by functions (or simply for general clarity). Finally, it declares and initializes a few global variables. The definitions and declarations appear in the ObjectARX header files.

NOTE If an application does not adhere to the conventions imposed by the definitions and declarations described in this chapter, it will be difficult to read and maintain at best; at worst, it will not communicate with AutoCAD correctly. Also, future versions of ObjectARX may involve changes to the header files. Therefore, do not substitute an integer constant for its symbolic code if such a code has been defined.

General Types and Definitions

The types and definitions described in this section provide consistency between applications and conformity with the requirements of AutoCAD. They also contribute to an application's legibility.

Real Numbers

Real values in AutoCAD are always double-precision floating-point values. ObjectARX preserves this standard by defining the special type `ads_real`, as follows:

```
typedef double ads_real;
```

Real values in an ObjectARX application are of the type `ads_real`.

Points

AutoCAD points are defined as the following array type:

```
typedef ads_real ads_point[3];
```

A point always includes three values. If the point is two-dimensional, the third element of the array can be ignored; it is safest to initialize it to 0.

ObjectARX defines the following point values:

```
#define X 0
#define Y 1
#define Z 2
```

Unlike simple data types (or point lists in AutoLISP), a point cannot be assigned with a single statement. To assign a pointer, you must copy the individual elements of the array, as shown in the following example:

```
newpt[X] = oldpt[X];
newpt[Y] = oldpt[Y];
newpt[Z] = oldpt[Z];
```

You can also copy a point value with the `ads_point_set()` macro. The result is the second argument to the macro.

The following sample code sets the point `to` equal to the point `from`:

```
ads_point to, from;

from[X] = from[Y] = 5.0; from[Z] = 0.0;
ads_point_set(from, to);
```

NOTE This macro, like the `ads_name_set()` macro, is defined differently, depending on whether or not the symbol `__STDC__` (for standard C) is defined. The standard C version of `ads_point_set()` requires that your program include `string.h`.

```
#include <string.h>
```

Because of the argument-passing conventions of the C language, points are passed by reference without the address (indirection) operator `&`. (C always

passes array arguments by reference, with a pointer to the first element of the array.)

The `acedOsnap()` library function takes a point as an argument, and returns a point as a result. It is declared as follows:

```
int acedOsnap(pt, mode, result)
ads_point pt;
char *mode;
ads_point result;
```

The `acedOsnap()` function behaves like the AutoLISP `osnap` function. It takes a point (`pt`) and some object snap modes (specified in the string mode), and returns the nearest point (in `result`). The `int` value that `acedOsnap()` returns is a status code that indicates success (`RTNORM`) or failure.

The following code fragment calls `acedOsnap()`:

```
int findendpoint(ads_point oldpt, ads_point newpt)
{
    ads_point ptres;
    int foundpt;

    foundpt = acedOsnap(oldpt, "end", ptres);
    if (foundpt == RTNORM) {
        ads_point_set(ptres, newpt);
    }
    return foundpt;
}
```

Because points are arrays, `oldpt` and `ptres` are automatically passed to `acedOsnap()` by reference (that is, as pointers to the first element of each array) rather than by value. The `acedOsnap()` function returns its result (as opposed to its status) by setting the value of the `newpt` argument.

ObjectARX defines a pointer to a point when a pointer is needed instead of an array type.

```
typedef ads_real *ads_pointp;
```

Transformation Matrices

The functions `acedDragGen()`, `acedGrVecs()`, `acedNEntSelP()`, and `acedXformSS()` multiply the input vectors by the transformation matrix defined as a 4x4 array of real values.

```
typedef ads_real ads_matrix[4][4];
```

The first three columns of the matrix specify scaling and rotation. The fourth column of the matrix is a translation vector. ObjectARX defines the symbol `T` to represent the coordinate of this vector, as follows:

```
#define T 3
```

The matrix can be expressed as follows:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

The following function initializes an identity matrix.

```
void ident_init(ads_matrix id)
{
    int i, j;

    for (i=0; i<=3; i++)
        for (j=0; j<=3; j++)
            id[i][j] = 0.0;
    for (i=0; i<=3; i++)
        id[i][i] = 1.0;
}
```

The functions that pass arguments of the `ads_matrix` type treat a point as a column vector of dimension 4. The point is expressed in homogeneous coordinates, where the fourth element of the point vector is a scale factor that is normally set to 1.0. The final row of the matrix has the nominal value of [0,0,0,1]; it is ignored by the functions that pass `ads_matrix` arguments. In this case, the following matrix multiplication results from the application of a transformation to a point:

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1.0 \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \bullet \begin{bmatrix} X \\ Y \\ Z \\ 1.0 \end{bmatrix}$$

This multiplication gives us the individual coordinates of the point as follows:

$$X' = M_{00}X + M_{01}Y + M_{02}Z + M_{03}(1.0)$$

$$Y' = M_{10}X + M_{11}Y + M_{12}Z + M_{13}(1.0)$$

$$Z' = M_{20}X + M_{21}Y + M_{22}Z + M_{23}(1.0)$$

As these equations show, the scale factor and the last row of the matrix have no effect and are ignored. This is known as an affine transformation.

NOTE To transform a vector rather than a point, do not add in the translation vector $M_3\ M_{13}\ M_{23}$ (from the fourth column of the transformation matrix).

The following function implements the previous equations to transform a single point:

```
void xformpt(xform, pt, newpt)
ads_matrix xform;
ads_point pt, newpt;
{
    int i, j;

    newpt[X] = newpt[Y] = newpt[Z] = 0.0;

    for (i=X; i<=Z; i++) {
        for (j=X; j<=Z; j++)
            newpt[i] += xform[i][j] * pt[j];
// Add the translation vector.
        newpt[i] += xform[i][T];
    }
}
```

The following figure summarizes some basic geometrical transformations. (The values in an `ads_matrix` are actually `ads_real`, but they are shown here as integers for readability and to conform to mathematical convention.)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & T_X \\ 0 & 1 & 0 & T_Y \\ 0 & 0 & 1 & T_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translation *scaling* *2D rotation (in the XY plane)*

The `acedXformSS()` function—unlike the `acedDragGen()`, `acedGrVecs()`, or `acedNEntSelP()` functions—requires the matrix to do uniform scaling. That is, in the transformation matrix that you pass to `acedXformSS()`, the elements in the scaling vector S_X S_Y S_Z must all be equal; in matrix notation, $M_{00} = M_{11} = M_{22}$. Three-dimensional rotation is a slightly different case, as shown in the following figure:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation in the XY plane *rotation in the YZ plane* *rotation in the XY plane*

For uniform rotations, the 3x3 submatrix delimited by [0,0] and [2,2] is orthonormal. That is, each row is a unit vector and is perpendicular to the other rows; the scalar (dot) product of two rows is zero. The columns are also unit vectors that are perpendicular to each other. The product of an orthonormal matrix and its transpose equals the identity matrix. Two complementary rotations have no net effect.

Complex transformations can be accomplished by combining (or composing) nonidentity values in a single matrix.

NOTE The `acedTablet()` function uses a 3x3 matrix to transform 2D points. The `acedNEntSel()` function uses a 4x3 transformation matrix that is similar to the 4x4 transformation matrix, but it treats the point as a row.

Entity and Selection Set Names

In AutoLISP, the names of entities and selection sets are pairs of long integers. ObjectARX preserves this standard by defining such names as an array type, as follows:

```
typedef long ads_name[2];
```

As with `ads_point` variables, `ads_name` variables are always passed by reference but must be assigned element by element.

You can also copy an entity or selection set name by calling the `ads_name_set()` macro. As with `ads_point_set()` and ObjectARX functions, the result is the second argument to the macro.

The following sample code sets the name `newname` to equal `oldname`.

```
ads_name oldname, newname;

if (acdbEntNext(NULL, oldname) == RTNORM)
ads_name_set(oldname, newname);
```

NOTE This macro, like the `ads_point_set()` macro, is defined differently, depending on whether or not the symbol `__STDC__` (which stands for standard C) is defined. The standard C version of `ads_name_set()` requires your program to include *string.h*.

The `ads_name_equal()` macro compares the names in the following example:

```
if (ads_name_equal(oldname, newname))
...
```

To assign a null value to a name, call the `ads_name_clear()` macro, and test for a null entity or selection set name with the macro `ads_name_nil()`.

The following sample code clears the `oldname` set in a previous example:

```
ads_name_clear(oldname);
```

And the following code tests whether the name is `NULL`:

```
if (ads_name_nil(oldname))
...
```

ObjectARX creates the following data type for situations that require a name to be a pointer rather than an array:

```
typedef long *ads_namep;
```

Useful Values

ObjectARX defines the following preprocessor directives:

```
#define TRUE 1
#define FALSE 0
#define EOS'\0' // String termination character
```

The `PAUSE` symbol, a string that contains a single backslash, is defined for the `acedCommand()` and `acedCmd()` functions, as follows:

```
#define PAUSE "\\" // Pause in command argument list
```

NOTE The ObjectARX library doesn't define the values `GOOD` and `BAD`, which appear as return values in the code samples throughout this guide (especially in error-handling code). You can define them if you want, or substitute a convention that you prefer.

Result Buffers and Type Codes

A general-purpose result buffer (`resbuf`) structure handles all of the AutoCAD data types. Type codes are defined to specify the data types in a result buffer.

Result-Buffer Lists

Result buffers can be combined in linked lists, described later in detail, and are therefore suitable for handling objects whose lengths can vary and objects that can contain a mixture of data types. Many ObjectARX functions return or accept either single result buffers (such as `acedSetVar()`) or result-buffer lists (such as `acdbEntGet()` and `acdbTblSearch()`).

struct resbuf

The following result-buffer structure, `resbuf`, is defined in conjunction with a union, `ads_u_val`, that accommodates the various AutoCAD and ObjectARX data types, as follows:

```
union ads_u_val {
    ads_real rreal;
    ads_real rpoint[3];
    short rint; // Must be declared short, not int.
    char *rstring;
    long rlname[2];
    long rlong;
    struct ads_binary rbinary;
};

struct resbuf {
    struct resbuf *rbnext; // Linked list pointer
    short restype;
    union ads_u_val resval;
};
```

NOTE The `long` integer field `resval.rlong` is like the binary data field `resval.rbinary`; both hold extended entity data.

The following figure shows the schematic form of a result-buffer list:

Result Type Codes Defined by ObjectARX

The `restype` field of a result buffer is a `short` integer code that indicates which type of value is stored in the `resval` field of the buffer. For results passed to and from ObjectARX functions, ObjectARX defines the result type codes listed in the following table:

Result type codes

Code	Description
RTNONE	No result value
RTREAL	Real (floating-point) value
RTPOINT	2D point (X and Y; Z == 0.0)
RTSHORT	Short (16-bit) integer
RTANG	Angle
RTSTR	String
RTENAME	Entity name
RTPICKS	Selection set name
RTORINT	Orientation
RT3DPOINT	3D point (X, Y, and Z)
RTLONG	Long (32-bit) integer
RTVOID	Void (blank) symbol
RTLB	List begin (for nested list)
RTLE	List end (for nested list)
RTDOTE	Dot (for dotted pair)
RTT	AutoLISP t (true)
RTNIL	AutoLISP nil
RTDXF0	Group code zero for DXF lists (used only with acutBuildList())

DXF Group Codes

Many ObjectARX functions return the type codes defined in the preceding table. However, in results from the functions that handle entities, the `restype` field contains DXF group codes, which are described in the *AutoCAD Customization Guide*. For example, in an entity list, a `restype` field of 10 indicates a point, while a `restype` of 41 indicates a real value.

AutoCAD drawings consist of structured containers for database objects having the following components:

- A unique handle that is always enabled and that persists for the lifetime of the drawing
- An optional xdata list
- An optional persistent reactor set
- An optional ownership pointer to an extension dictionary, which owns other database objects placed in it by the application

Database objects are objects without layer, linetype, color, or any other geometric or graphical properties, and entities are derived from objects and have geometric and graphical properties.

Because DXF codes are always less than 2,000 and the result type codes are always greater, an application can easily determine when a result-buffer list contains result values (as returned by `acedGetArgs()`, for example) or contains entity definition data (as returned by `acdbEntGet()` and other entity functions).

The following figure shows the result-buffer format of a circle retrieved by `acdbEntGet()`:

The following sample code fragment shows a function, `dxftype()`, that is passed a DXF group code and the associated entity, and returns the corresponding type code. The type code indicates what data type can represent the data: `RTREAL` indicates a double-precision floating-point value, `RT3DPOINT` indicates an `ads_point`, and so on. The kind of entity (for example, a normal entity such as a circle, a block definition, or a table entry such as a viewport) is indicated by the type definitions that accompany this function:

```
#define ET_NORM 1 // Normal entity
#define ET_TBL  2 // Table
#define ET_VPORT  3 // Table numbers
#define ET_LTYPE  4
#define ET_LAYER  5
#define ET_STYLE  6
#define ET_VIEW   7
#define ET_UCS    8
#define ET_BLOCK  9

// Get basic C-language type from AutoCAD DXF group code (RTREAL,
// RTANG are doubles, RTPOINT double[2], RT3DPOINT double[3],
// RTENAME long[2]). The etype argument is one of the ET_
// definitions.
//
// Returns RTNONE if grpcode isn't one of the known group codes.
// Also, sets "inxdata" argument to TRUE if DXF group is in XDATA.

short dxftype(short grpcode, short etype, int *inxdata)
{
    short rbtype = RTNONE;

    *inxdata = FALSE;
    if (grpcode >= 1000) {  // Extended data (XDATA) groups
        *inxdata = TRUE;
        if (grpcode == 1071)
            rbtype = RTLONG; // Special XDATA case
        else
            grpcode %= 1000; // All other XDATA groups match.
    } // regular DXF code ranges
    if (grpcode <= 49) {
        if (grpcode >= 20) // 20 to 49
            rbtype = RTREAL;
        else if (grpcode >= 10) { // 10 to 19
            if (etype == ET_VIEW) // Special table cases
                rbtype = RTPOINT;
            else if (etype == ET_VPORT && grpcode <= 15)
                rbtype = RTPOINT;
            else // Normal point
               rbtype = RT3DPOINT; // 10: start point, 11: endpoint
        }
        else if (grpcode >= 0) // 0 to 9
            rbtype = RTSTR; // Group 1004 in XDATA is binary
        else if (grpcode >= -2)
```

```
// -1 = start of normal entity -2 = sequence end, etc.
            rbtype = RTENAME;
        else if (grpcode == -3)
            rbtype = RTSHORT; // Extended data (XDATA) sentinel
    }
    else {
        if (grpcode <= 59) // 50 to 59
            rbtype = RTANG; // double
        else if (grpcode <= 79) // 60 to 79
            rbtype = RTSHORT;
        else if (grpcode < 210)
            ;
        else if (grpcode <= 239) // 210 to 239
            rbtype = RT3DPOINT;
        else if (grpcode == 999) // Comment
            rbtype = RTSTR;
    }
    return rbtype;
}
```

An application obtains a result-buffer list (called `rb`), representing an entry in the viewport symbol table, and the following C statement calls `dxftype()`:

```
ctype = dxftype(rb->restype, ET_VPORT, &inxdata);
```

Suppose `rb->restype` equals 10. Then `dxftype()` returns `RTPOINT`, indicating that the entity is a two-dimensional point whose coordinates (of the type `ads_real`) are in `rb->resval.rpoint[X]` and `rb->resval.rpoint[Y]`.

ObjectARX Function Result Type Codes

The following result type codes are the status codes returned by most ObjectARX global functions to indicate success, failure, or special conditions (such as user cancellation):

Library function result codes

Code	Description
RTNORM	User entered a valid value
RTERROR	The function call failed
RTCAN	User entered ESC
RTREJ	AutoCAD rejected the request as invalid
RTFAIL	AutoLISP communication failed
RTKWORD	User entered a keyboard or arbitrary text

The meanings of these codes, summarized in the table, are as follows:

RTNORM	The library function succeeded.
RTERROR	The library function did not succeed; it encountered a recoverable error.

The `RTERROR` condition is exclusive of the following special cases:

RTCAN	The AutoCAD user entered ESC to cancel the request. This code is returned by the user-input (`acedGetxxx`) functions and by the following functions: `acedCommand`, `acedCmd`, `acedEntSel`, `acedNEntSelP`, `acedNEntSel`, and `acedSSGet`.
RTREJ	AutoCAD rejected the operation as invalid. The operation request may be incorrectly formed, such as an invalid `acdbEntMod()` call, or it simply may not be valid for the current drawing.
RTFAIL	The link with AutoLISP failed. This is a `fatal` error that probably means AutoLISP is no longer running correctly. If it detects this error, the application should quit. (Not all applications check for this code, because the conditions that can lead to it are likely to hang AutoCAD, anyway.)
RTKWORD	The AutoCAD user entered a keyword or arbitrary input instead of another value (such as a point). The user-input `acedGetxxx()` functions, as well as `acedEntSel`, `acedEntSelP`, `acedNEntSel`, and `acedDragGen`, return this result code.

NOTE Not all ObjectARX global functions return these status codes; some return values directly. Also, the user-input (`acedGetxxx`, `acedEntSel`, `acedEntSelP`, `acedNEntSel`, and `acedDragGen`) functions can return the `RTNONE` result type code, and `acedDragGen()` indicates arbitrary input by returning `RTSTR` instead of `RTKWORD`.

User-Input Control Bit Codes

The user-input control bit codes listed in the following table are passed as the first argument to the `acedInitGet()` function to control the behavior of user-

input functions `acedGetxxx`, `acedEntSel`, `acedNEntSelP`, `acedNEntSel`, and `acedDragGen`:

User-input control bit codes	
Code	**Description**
RSG_NONULL	Disallow null input
RSG_NOZERO	Disallow zero values
RSG_NONEG	Disallow negative values
RSG_NOLIM	Do not check drawing limits, even if LIMCHECK is on
RSG_DASH	Use dashed lines when drawing rubber-band line or box
RSG_2D	Ignore *Z* coordinate of 3D points (acedGetDist() only)
RSG_OTHER	Allow arbitrary input (whatever the user types)

Lists and Other Dynamically Allocated Data

The resbuf structure includes a pointer field, `rbnext`, for linking result buffers into a list. Result buffers can be allocated statically by declaring them in the application. You do this when only a single result buffer is used (for example, by `acedGetVar()` and `acedSetVar()`) or when only a short list is needed. But longer lists are easier to handle by allocating them dynamically, and lists returned by ObjectARX functions are always allocated dynamically. One of the most frequently used functions that returns a linked list is `acedGetArgs()`.

"Evaluating External Functions" on page 526 shows the AutoLISP calling format of an external subroutine that takes arguments of three distinct types: a string, an integer, and a real value:

```
(doit pstr iarg rarg)
```

The following code segment shows how to implement a function with such a calling sequence. The sample function checks that the argument list is correct and saves the values locally before operating on them (operations are not

shown). The example assumes that a previous call to `acedDefun()` has assigned the external subroutine a function code of 0, and that all functions defined by this application take at least one argument:

```
// Execute a defined function.

int dofun()
{
    struct resbuf *rb;
    char str[64];
    int ival, val;
    ads_real rval;
    ads_point pt;

// Get the function code.
    if ((val = acedGetFuncode()) == RTERROR)
        return BAD; // Indicate failure.
    // Get the arguments passed in with the function.
    if ((rb = acedGetArgs()) == NULL)
        return BAD;

    switch (val) { // Which function is called?

    case 0: // (doit)
        if (rb->restype != RTSTR) {
            acutPrintf("\nDOIT called with %d type.",
                rb->restype);
            acutPrintf("\nExpected a string.");
            return BAD;
        }
// Save the value in local string.
        strcpy(str, rb->resval.rstring);

// Advance to the next result buffer.
        rb = rb->rbnext;
        if (rb == NULL) {
            acutPrintf("\nDOIT: Insufficient number of
                arguments.");
            return BAD;
        }

        if (rb->restype != RTSHORT) {
            acutPrintf("\nDOIT called with %d type.",
                rb->restype);
            acutPrintf("\nExpected a short integer.");
            return BAD;
        }

// Save the value in local variable.
        ival = rb->resval.rint;
```

```
// Advance to the last argument.
        rb = rb->rbnext;
        if (rb == NULL) {
            acutPrintf("\nDOIT: Insufficient number of
                arguments.");
            return BAD;
        }

        if (rb->restype != RTREAL) {
            acutPrintf("\nDOIT called with %d type.",
                rb->restype);
            acutPrintf("\nExpected a real.");
            return BAD;
        }

// Save the value in local variable.
        rval = rb->resval.rreal;

// Check that it was the last argument.
        if (rb->rbnext != NULL) {
                acutPrintf("\nDOIT: Too many arguments.");
            return BAD;
        }

// Operate on the three arguments.
        . . .
        return GOOD;  // Indicate success
        break;

    case 1:
// Execute other functions.
        . . .
    }
}
```

NOTE This example is exceptional in one respect: `acedGetArgs()` is the only ObjectARX global function that returns a linked list that the application does not have to explicitly release. The following section describes the usual way of managing the memory needed for lists.

Result-Buffer Memory Management

The main difference between result-buffer lists and comparable AutoLISP result lists is that an ObjectARX application must explicitly manage the lists that it creates and uses. Whether an application creates a list or has one passed to it, it is the application's responsibility to release the result buffers that it allocates. ObjectARX has no automatic garbage collection as AutoLISP does. The application must call the library function `acutRelRb()` to release dynamically allocated result buffers when the application is finished with them.

The `acutRelRb()` function releases the entire list that follows the specified result buffer, including the specified (head) buffer itself and any string values that the buffers in the list point to. To release a string without removing the buffer itself, or to release a string belonging to a static result buffer, the application must call the standard C library function `free()`.

WARNING! Do not write data to a dynamic location that hasn't been allocated with direct calls to `malloc()` or with the ObjectARX library (including `acutNewRb()`). This can corrupt data in memory. Conversely, calling `free()` or `acutRelRb()` to release data that was allocated statically—in a static or automatic variable declaration—also can corrupt memory. Inserting a statically allocated variable, such as a string, into a result-buffer list causes your program to fail when you release the list with `acutRelRb()`.

Sample calls to `acutRelRb()` appear in several of the code examples in the following sections.

List Creation and Deletion

An ObjectARX application can dynamically allocate a single result buffer by calling `acutNewRb()`. The call to `acutNewRb()` must specify the type of buffer to allocate; `acutNewRb()` automatically initializes the buffer's `restype` field to contain the specified type code.

The following sample code fragment allocates a result buffer to contain a three-dimensional point and then initializes the point value:

```
struct resbuf *head;

if ((head=acutNewRb(RT3DPOINT)) == NULL) {
    acdbFail("Unable to allocate buffer\n");
    return BAD;
}

head->resval.rpoint[X] = 15.0;
head->resval.rpoint[Y] = 16.0;
head->resval.rpoint[Z] = 11.18;
```

If the new result buffer is to contain a string, the application must explicitly allocate memory to contain the string:

```
struct resbuf *head;

if ((head=acutNewRb(RTSTR)) == NULL) {
    acdbFail("Unable to allocate buffer\n");
    return BAD;
}
```

```
if ((head->resval.rstring = malloc(14)) == NULL) {
    acdbFail("Unable to allocate string\n");
    return BAD;
}

strcpy(head->resval.rstring, "Hello, there.");
```

Memory allocated for strings that are linked to a dynamic list is released when the list is released, so the following call releases all memory allocated in the previous example:

```
acutRelRb(head);
```

To release the string without releasing the buffer, call `free()` and set the string pointer to `NULL` as shown in the following example:

```
free(head->resval.rstring);
head->resval.rstring = NULL;
```

Setting `resval.rstring` to `NULL` prevents a subsequent call to `acutRelRb()` from trying to release the string a second time.

If the elements of a list are known beforehand, a quicker way to construct it is to call `acutBuildList()`, which takes a variable number of argument pairs (with exceptions such as `RTLB`, `RTLE`, `-3`, and others) and returns a pointer to a list of result buffers that contains the specified types and values, linked together in the order in which they were passed to `acutBuildList()`. This function allocates memory as required and initializes all values. The last argument to `acutBuildList()` must be a single argument whose value is either zero or `RTNONE`.

The following sample code fragment constructs a list that consists of three result buffers. These contain a real value, a string, and a point, in that order:

```
struct resbuf *result;
ads_point pt1 = {1.0, 2.0, 5.1};

result = acutBuildList(
    RTREAL, 3.5,
    RTSTR, "Hello, there.",
    RT3DPOINT, pt1,
    0 );
```

If it cannot construct the list, `acutBuildList()` returns `NULL`; otherwise, it allocates space to contain the list. This list must be released by a subsequent call to `acutRelRb()`:

```
if (result != NULL)
    acutRelRb(result);
```

AutoLISP Lists

The `acutBuildList()` function is called in conjunction with `acedRetList()`, which returns a list structure to AutoLISP.

The following sample code fragment passes a list of four points:

```
struct resbuf *res_list;
ads_point ptarray[4];

// Initialize the point values here.
.
.
.

res_list = acutBuildList(
    RT3DPOINT, ptarray[0],
    RT3DPOINT, ptarray[1],
    RT3DPOINT, ptarray[2],
    RT3DPOINT, ptarray[3], 0);

if (res_list == NULL) {
    acdbFail("Couldn't create list\n");
    return BAD;
}

acedRetList(res_list);
acutRelRb(res_list);
```

Dotted pairs and nested lists can be returned to AutoLISP by calling `acutBuildList()` to build a list created with the special list-construction type codes. These codes are needed only for complex lists. For ordinary (that is, one-dimensional) lists, `acedRetList()` can be passed a simple list of result buffers, as shown in the previous example.

NOTE A list returned to AutoLISP by `acedRetList()` can include only the following result type codes: `RTREAL`, `RTPOINT`, `RTSHORT`, `RTANG`, `RTSTR`, `RTENAME`, `RTPICKS`, `RTORINT`, `RT3DPOINT`, `RTLB`, `RTLE`, `RTDOTE`, `RTNIL`, and `RTT`. (Although there is an `RTNIL` return code, if you are returning only a `nil` list, you can call `acedRetNil()`). It can contain result types of `RTLONG` if the list is being returned to another ObjectARX application.

Use of the list-construction type codes is simple. In the `acutBuildList()` call, a nested list is preceded by the result type code `RTLB` (for *L*ist *B*egin) and is followed by the result type code `RTLE` (for *L*ist *E*nd). A dotted pair can also be constructed. Dotted pairs also begin with `RTLB` and end with `RTLE`; the dot is indicated by the result type code `RTDOTE`, and appears between the two members of the pair.

NOTE This is a change from earlier versions. Applications that receive a dotted pair from AutoLISP no longer have to modify the format of the dotted pair before returning it with `acedRetList()`. (The earlier order, with `RTDOTE` at the end, is still supported.)

WARNING! The `acutBuildList()` function does not check for a well-formed AutoLISP list. For example, if the `RTLB` and `RTLE` codes are not balanced, this error is not detected. If the list is not well formed, AutoLISP can fail. Omitting the `RTLE` code is guaranteed to be a fatal error.

The following sample code fragment constructs a nested list to return to AutoLISP:

```
res_list = acutBuildList(
    RTLB, // Begin sublist.
    RTSHORT, 1,
    RTSHORT, 2,
    RTSHORT, 3,
    RTLE, // End sublist.
    RTSHORT, 4,
    RTSHORT, 5,
    0);

if (res_list == NULL) {
    acdbFail("Couldn't create list\n");
    return BAD;
}

acedRetList(res_list);
acutRelRb(res_list);
```

The list that this example returns to AutoLISP has the following form:

((1 2 3) 4 5)

The following code fragment constructs a dotted pair to return to AutoLISP:

```
res_list = acutBuildList(
    RTLB, // Begin dotted pair.
    RTSTR, "Sample",
    RTDOTE,
    RTSTR, "Strings",
    RTLE, // End dotted pair.
    0);

if (res_list == NULL) {
    acdbFail("Couldn't create list\n");
    return BAD;
}
```

```
acedRetList(res_list);
acutRelRb(res_list);
```

The list that this example returns to AutoLISP has the following form:

((“Sample” . “Strings”))

NOTE In AutoLISP, dotted pairs associate DXF group codes and values. In an ObjectARX application this is unnecessary, because a single result buffer contains both the group code (in its `restype` field) and the value (in its `resval` field). While ObjectARX provides the list-construction type codes as a convenience, most ObjectARX applications do not require them.

Entity Lists with DXF Codes in ObjectARX

As previously mentioned, lists with DXF group codes represent AutoCAD entities. The `acutBuildList()` function constructs such lists. To construct an entity, call both `acutBuildList()` and `acdbEntMake()`.

NOTE Entity definitions begin with a zero (`0`) group that describes the entity type. Because lists passed to `acutBuildList()` are terminated with `0` (or `RTNONE`), this creates a conflict. The special result type code `RTDXF0` resolves the conflict. Construct the zero group in DXF lists passed to `acutBuildList()` with `RTDXF0`. If you attempt to substitute a literal zero for `RTDXF0`, `acutBuildList()` truncates the list.

The following sample code fragment creates a DXF list that describes a circle and then passes the new entity to `acdbEntMake()`. The circle is centered at (4,4), has a radius of 1, and is colored red:

```
struct resbuf *newent;
ads_point center = {4.0, 4.0, 0.0};

newent = acutBuildList(
    RTDXF0, "CIRCLE",
    62, 1, // 1 == red
    10, center,
    40, 1.0, // Radius
    0 );

if (acdbEntMake(newent) != RTNORM) {
    acdbFail("Error making circle entity\n");
    return BAD;
}
```

Command and Function Invocation Lists

Finally, `acutBuildList()` is called in conjunction with `acedCmd()`, which takes a result-buffer list to invoke AutoCAD commands, and with `acedInvoke()`, which invokes an external function from a different ObjectARX application.

The following sample code fragment calls `acutBuildList()` and `acedInvoke()` to invoke the RESET command defined by the sample application *gravity.c*:

```
struct resbuf *callist, *results = NULL;

callist = acutBuildList(RTSTR, "c:reset", 0);
if (acedInvoke(callist, &results) == RTERROR)
    acdbFail("Cannot run RESET -- GRAVITY program may not
    be loaded\n");

acutRelRb(callist);
acutRelRb(results);
```

Extended Data Exclusive Data Types

Extended data (xdata) can include binary data, organized into variable-length chunks. These are handled by the `ads_binary` structure, as follows:

```
struct ads_binary { // Binary data chunk structure
    short clen; // Length of chunk in bytes
    char *buf; // Binary data
};
```

The value of the `clen` field must be in the range of 0 to 127. If an application requires more than 127 bytes of binary data, it must organize the data into multiple chunks.

With Release 13, the DXF representation of a symbol table can include extended entity data. Xdata is returned as a handle.

NOTE There is no mechanism for returning binary data to AutoLISP. Binary chunks can be passed to other external functions by means of `acedInvoke()`, but only when they belong to groups (1004) within an entity's extended data. You cannot pass isolated binary chunks.

Xdata can also include `long` integers. The `ads_u_val` union of the `resval` field of a result buffer includes both an `ads_binary` and a `long` member for handling extended entity data.

NOTE There is no mechanism for returning a `long` integer to AutoLISP. Long integers can be passed to other external functions by means of `acedInvoke()`, but only when they belong to groups (1071) within an entity's extended data. In AutoLISP, 1071 groups are maintained as real values.

Text String Globalization Issues

AutoCAD Release 13 was enhanced with localization support to make AutoCAD more suitable for international customers. With this support, an AutoCAD user can enter commands in local non-English languages, and the display shows messages in the local language. The support for multiple-language character sets involves out-of-code-page characters.

Sometimes system code page strings in a *.dwg* file have out-of-code-page characters to display messages in another language. These characters have no normal representation in the character set of the native system. The "`\U+xxxx`" and "`\M+xxxx`" escape sequences represent these special characters in the system code page strings. The `xxxx` is a sequence of four hexadecimal digits that specify either the Unicode (single-character encoding) identifier or Multibyte Interchange Format (MIF) of the encoded character.

As part of Autodesk's globalization effort, the following preexisting ObjectARX functions have been changed to improve the handling of drawings created with various language versions of AutoCAD:

acdbXdSize	Returns the number of bytes of memory needed for a list of extended entity data.
acdbXdRoom	Returns the number of bytes of memory that an entity has available for extended data.

These functions count out-of-code-page characters differently.

The `acdbXdSize()` and `acdbXdRoom()` functions now recognize "`\U+xxxx`" as 1 byte, but other ObjectARX functions recognize "`\U+xxxx`" as 7 bytes. The Asian version of AutoCAD recognizes "`\M+xxxx`" as 2 bytes.

NOTE ObjectARX applications that make explicit assumptions about the limit of the string length of symbol table names and TEXT entities are affected by out-of-code-page characters.

Input Point Processing

In This Chapter

21

- Custom Object Snap Modes
- Input Point Management

ObjectARX allows applications to customize input point processing. The application can associate new object snap points and AutoSnap alignment lines with custom and existing entities, and can monitor the input point process and modify the input points. This chapter discusses these topics.

Custom Object Snap Modes

ObjectARX provides the ability to create custom object snap modes. These modes allow applications to associate new object snap points and AutoSnap alignment lines with custom and existing entities. To create a custom object snap mode, you must do the following:

- Create and register a custom object snap mode.
- Create protocol extension classes to perform the input point processing.
- Create a custom glyph.

The following sections discuss these subjects in more detail.

Creating and Registering a Custom Object Snap Mode

To create a custom object snap mode, you must derive a subclass from `AcDbCustomOsnapMode` and register it with the custom object snap manager.

Using the Custom Object Snap Manager

The custom object snap manager handles the registration of custom object snap modes. It can be used to add, remove, activate, and deactivate custom object snap modes. The custom object snap manager also can be used to query whether a specified object snap mode is registered and active.

There is a single custom object snap manager for the entire application. Once registered, a custom object snap mode can be applied in any open document. The following global function can be used to access the custom object snap manager:

```
AcDbCustomOsnapManager *
acdbCustomOsnapManager() const;
```

Typically, custom object snap modes are registered when the application is first loaded and are removed when the application is unloaded, although they can be registered and removed at any time.

Creating Custom Object Snap Modes

Custom object snap modes allow the developer to set the following attributes:

- Keyword

  The custom object snap keyword that the user types in to activate the object snap. Both local and global keywords must be specified.
- Protocol extension class

  A pointer to the class that performs the per-entity processing of the object snap mode.
- Glyph

  The custom glyph for the object snap mode.
- ToolTip string

  The default ToolTip string for the custom object snap mode.

For more information on setting these attributes, see the *ObjectARX Reference*.

A custom object snap mode is defined by registering an instance of the `AcDbCustomOsnapMode` class with the custom object snap manager, described in the previous section.

When a custom object snap mode is used, AutoCAD takes the class object returned by `AcDbCustomOsnapMode::entityOsnapClass()`, looks up the corresponding protocol extension object for the picked entity, and invokes `AcDbCustomOsnapInfo::getOsnapInfo()` to obtain the points or lines associated with that entity and object snap mode. If the final candidate point is associated with that object snap mode, AutoCAD displays the glyph object from the instance returned by `AcDbCustomOsnapMode::glyph()` and the ToolTip string returned by `AcDbCustomOsnapMode::tooltipString()`.

Creating Protocol Extension Classes

To create a custom object snap mode you must create protocol extension classes to handle the per-entity input point processing. The class `AcDbCustomOsnapInfo` defines the protocol that every custom object snap mode must implement for relevant entities. This base class contains the function `getOsnapInfo()`, which performs the input point processing on an entity:

```
class AcDbCustomOsnapInfo : public AcRxObject {
public:
ACRX_DECLARE_MEMBERS(AcDbCustomOsnapInfo);
```

```
virtual Acad::ErrorStatus
getOsnapInfo(
    AcDbEntity* pickedObject,
    int gsSelectionMark,
    const AcGePoint3d& pickPoint,
    const AcGePoint3d& lastPoint,
    const AcGeMatrix3d& viewXform,
    AcArray<AcGePoint3d>& snapPoints,
    AcArray<int>& geomIdsForPts,
    AcArray<AcGeLine3d>& snapLines,
    AcArray<int>& geomIdsForLines);
};
```

To create protocol extension classes for custom object snap modes

1. Define an abstract base protocol extension class derived from `AcDbCustomOsnapInfo`.

   For example, if your custom class is called `AcmeSocketInfo`, define it as follows:

```
class AcmeSocketInfo : public AcDbCustomOsnapInfo{
public:
ACRX_DECLARE_MEMBERS(AcDbSocketInfo);

virtual Acad::ErrorStatus
getOsnapInfo(
    AcDbEntity* pickedObject,
    int gsSelectionMark,
    const AcGePoint3d& pickPoint,
    const AcGePoint3d& lastPoint,
    const AcGeMatrix3d& viewXform,
    AcArray<AcGePoint3d>& snapPoints,
    AcArray<int>& geomIdsForPts,
    AcArray<AcGeLine3d>& snapLines,
    AcArray<int>& geomIdsForLines);
};
ACRX_NO_CONS_DEFINE_MEMBERS(AcmeSocketInfo, AcDbCustomOsnapInfo);
```

2. Initialize the base protocol extension class and add it to the runtime class hierarchy.

   For example, add the following lines to your `acrxEntryPoint()` function:

```
AcmeSocketInfo::rxInit();
acrxBuildClassHierarchy();
```

3. For every relevant entity class, derive a protocol extension class from the base class.

   For example, you might derive a class called `AcmeSocketForLines` that implements `getOsnapInfo()` to handle the input point processing for lines.

NOTE A default implementation should be associated with `AcDbEntity` for each registered class derived from `AcDbCustomOsnapInfo`.

4 Create an instance of each protocol extension object and add the objects to the appropriate `AcRxClass` descriptor objects using the `addX()` function.

For example:

```
pSocketForLine = new AcmeSocketForLine;
AcDbLine::desc()->addX(AcmeSocketInfo::desc(), pSocketForLine);
```

Creating a Custom Glyph

You can create a custom glyph by deriving from `AcGiGlyph` and registering the glyph with your custom object snap mode. Two functions need to be implemented in your derived class: `setLocation()` and `viewportDraw()`. The `setLocation()` function sets the location of the glyph to be drawn, and the `viewportDraw()` function draws the glyph.

There are a few requirements for the graphics used in `viewportDraw()` that must be observed. The graphics should be display-aligned, and they should not be affected by entity orientation, the current UCS, or the current view transform. Additionally, the graphics should be scaled to fit the current AutoSnap marker size, which can be determined by using the function `acdbCustomOsnapManager()->osnapGlyphSize()`.

NOTE If you return a `NULL` pointer instead of a custom glyph, AutoCAD will not draw any glyph for the object snap mode.

Custom Object Snap Mode Example

The following example demonstrates the creation of custom object snap modes:

```
#include "rxobject.h"
#include "ads.h"
#include "adslib.h"
#include "dbmain.h"
#include "dbents.h"
#include "dbosnap.h"
#include "acedinpt.h"
```

```
// Socket Osnap mode protocol extension class.
//
class AcmeSocketInfo : public AcDbCustomOsnapInfo {
public:
ACRX_DECLARE_MEMBERS(AcmeSocketInfo);

virtual Acad::ErrorStatus
getOsnapInfo(
    AcDbEntity*           pickedObject,
    int                   gsSelectionMark,
    const AcGePoint3d&    pickPoint,
    const AcGePoint3d&    lastPoint,
    const AcGeMatrix3d&   viewXform,
    AcArray<AcGePoint3d>& snapPoints,
    AcArray<int>&         geomIdsForPts,
    AcArray<AcGeCurve3d>& snapCurves,
    AcArray<int>&         geomIdsForLines)
};

// This class is registered with AcRx, to be used by the host
// application to look up entity class-specific implementations.
//
ACRX_NO_CONS_DEFINE_MEMBERS(AcmeSocketInfo,AcDbCustomOsnapInfo);

Acad::ErrorStatus
AcmeSocketInfo::getOsnapInfo(
    AcDbEntity*,
    int,
    const AcGePoint3d&,
    const AcGePoint3d&,
    const AcGePoint3d&,
    AcArray<AcGePoint3d>& snapPoints,
    AcArray<int>&         geomIdsForPts,
    AcArray<AcGeCurve3d>& snapCurves,
    AcArray<int>&         geomIdsForLines)
{
    // Associate with AcDbEntity to define default behavior.
    //
    snapPoints.setLogicalLength(0);
    geomIdsForPts.setLogicalLength(0);
    snapLiness.setLogicalLength(0);
    geomIdsForLines.setLogicalLength(0);
}
```

```
// Socket Osnap mode protocol extension object for AcDbLine.
//
class AcmeSocketForLine : public AcmeSocketInfo {
public:

virtual Acad::ErrorStatus
getOsnapInfo(
    AcDbEntity*           pickedObject,
    int                   gsSelectionMark,
    const AcGePoint3d&    pickPoint,
    const AcGePoint3d&    lastPoint,
    const AcGeMatrix3d&   viewXform,
    AcArray<AcGePoint3d>& snapPoints,
    AcArray<int>&         geomIdsForPts,
    AcArray<AcGeCurve3d>& snapCurves,
    AcArray<int>&         geomIdsForLines)
};

Acad::ErrorStatus
AcmeSocketForLine::getOsnapInfo(
    AcDbEntity*          pickedObject,
    int,
    const AcGePoint3d&    pickPoint,
    const AcGePoint3d&,
    const AcGeMatrix3d&   viewXform,
    AcArray<AcGePoint3d>& snapPoints,
    AcArray<int>&         geomIdsForPts,
    AcArray<AcGeCurve3d>& snapCurves,
    AcArray<int>&         geomIdsForLines)
{
    // Protocol extension ensures that the following assertion
    // is always true, but check in non-production versions
    // just to be safe.
    //
    ASSERT(pickedObject->isKindOf(AcDbLine::desc()));

    // In production, a hard cast is fastest.
    AcDbLine* lineEnt = (AcDbLine*)pickedObject;

    // Do computation using AcDbLine protocol, pickPoint, and
    // viewXform. For example, if you want to find the closest
    // socket to the pick point and return just that, set
    // snapPoints and geomIdsForPts accordingly.

    // But this isn't an AutoSnap mode...
    //
    snapLiness.setLogicalLength(0);
    geomIdsForLines.setLogicalLength(0);
}
```

```
// Actual protocol extension objects
//
static AcmeSocketInfo*          pDefaultSocketInfo = NULL;
static AcmeSocketForLine*       pSocketForLine = NULL;

// "SOCket" Osnap mode glyph object
//
class AcmeSocketGlyph : public AcGiGlyph
{
public:

virtual Acad::ErrorStatus
setLocation(const AcGePoint3d& dcsPoint);

virtual void
viewportDraw(AcGiViewportDraw* vportDrawContext);

private:
    AcGePoint3d mCurDcsLoc;

};

Acad::ErrorStatus
AcmeSocketGlyph::setLocation(const AcGePoint3d& dcsPoint)
{
    mCurDCSLoc = dcsPoint;
}

// These variables are extremely transient, and are
// made static to save constructor/destructor cost.
static AcGePoint2d&   sPixelArea;
AcArray<AcGePoint3d>  sSegmentPoints[2];

void
AcmeSocketGlyph::viewportDraw(AcGiViewportDraw* vportDrawContext)
{
    // Taking mCurDCSLoc, the pixel size, and the AutoSnap
    // marker size into account, plus anything else, such as socket
    // orientation, draw the glyph.

    // If this ASSERT fails, then the pixel size is really position-
    // dependent.
    //
    ASSERT(!vportDrawContext->viewport()->isPerspective());

    vportDrawContext->viewport()->
        getNumPixelsInUnitSquare(
            AcGePoint3d::kOrigin, pixelArea);

    double halfGlyphSizeInDCS =
      acdbCustomOsnapManager->osnapGlyphSize() * pixelArea.x / 2.0;
```

```
    // Draw an asterisk with 4 segments.
    //
    sSegmentPoints[0].set(
        mCurDCSLoc.x-halfGlyphSizeInDCS,
        mCurDCSLoc.y-halfGlyphSizeInDCS, 0.0);
    sSegmentPoints[1].set(
        mCurDCSLoc.x+halfGlyphSizeInDCS,
        mCurDCSLoc.y+halfGlyphSizeInDCS, 0.0);
    vportDrawContext->geometry().polylineDc(
        2, &(sSegmentPoints[0]));
    sSegmentPoints[0].set(
        mCurDCSLoc.x-halfGlyphSizeInDCS,
        mCurDCSLoc.y+halfGlyphSizeInDCS, 0.0);
    sSegmentPoints[1].set(
        mCurDCSLoc.x+halfGlyphSizeInDCS,
        mCurDCSLoc.y-halfGlyphSizeInDCS, 0.0);
    vportDrawContext->geometry().polylineDc(
        2, &(sSegmentPoints[0]));
    sSegmentPoints[0].set(
        mCurDCSLoc.x-halfGlyphSizeInDCS,
        mCurDCSLoc.y, 0.0);
    sSegmentPoints[1].set(
        mCurDCSLoc.x+halfGlyphSizeInDCS,
        mCurDCSLoc.y, 0.0);
    vportDrawContext->geometry().polylineDc(
        2, &(sSegmentPoints[0]));
    sSegmentPoints[0].set(
        mCurDCSLoc.x,
        mCurDCSLoc.y-halfGlyphSizeInDCS, 0.0);
    sSegmentPoints[1].set(
        mCurDCSLoc.x,
        mCurDCSLoc.y+halfGlyphSizeInDCS, 0.0);
    vportDrawContext->geometry().polylineDc(
        2, &(sSegmentPoints[0]));
};

AcmeSocketGlyph* pSocketGlyph = NULL;
```

```
// Master object for the socket custom Osnap mode.
//
class AcmeSocketMode : public AcDbCustomOsnapMode {
public:

virtual const char*
localModeString() const  {return "SOCket"};

virtual const char*
globalModeString() const {return "SOCket"};

virtual const AcRxClass*
entityOsnapClass() const {return AcmeSocketInfo::desc());

virtual AcGiGlyph*
glyph() const {return pSocketGlyph;);

virtual const char*
tooltipString() const  {return "Socket to Me?" };
};

static AcmeSocketMode* pSocketMode = NULL;


/* ================ ObjectARX application interface ============ */

AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void*) {
    switch(msg) {

    case AcRx::kInitAppMsg:

        // Register the class.
        //
        AcmeSocketInfo::rxInit();
        acrxBuildClassHierarchy();

        pDefaultSocketInfo = new AcmeSocketInfo;
        AcDbEntity::desc()->addX(AcmeSocketInfo::desc(),
            pDefaultSocketInfo);
        pSocketForLine = new AcmeSocketForLine;
        AcDbLine::desc()->addX(AcmeSocketInfo::desc(),
            pSocketForLine);
    };

        // Create the glyph object to be returned by the socket
        // mode object.
        //
        pSocketGlyph = new AcmeSocketGlyph;

        // Create and register the custom Osnap Mode
        pSocketMode = new AcmeSocketMode;
        acdbCustomOsnapManager->addCustomOsnapMode(pSocketMode);
```

```
        // The SOCket Osnap mode is now plugged in and ready to
        // use.
        //
        break;

    case AcRx::kUnloadAppMsg:
        // Clean up.
       acdbCustomOsnapManager->removeCustomOsnapMode(pSocketMode);
        delete pSocketMode;

        // Unregister, then delete the protocol extension object.
        //
        AcDbEntity::desc()->delX(AcmeSocketInfo::desc());
        delete pDefaultSocketInfo;

        AcDbLine::desc()->delX(AcmeSocketInfo::desc());
        delete pSocketForLine;

        // Remove the protocol extension class definition.
        //
        acrxClassDictionary->remove("AcmeSocketInfo");
        break;

    default:
        // Between the initialization and termination of the
        // application, all registered objects will be directly
        // invoked as needed.  No commands or AutoLISP
        // expressions are necessary.
        //
        break;
    }
    return AcRx::kRetOK;
}
```

Input Point Management

Input points are acquired by running input events through a series of computations. These computations are referred to as filters, and they can be customized by the developer. ObjectARX provides APIs to monitor the input point process and modify the input points. The following sections discuss these topics.

Input Point Manager

ObjectARX provides an input point manager class, `AcEdInputPointManager`. One input point manager is instantiated for each active document in AutoCAD.

The following function returns the input point manager for a document:

```
virtual AcEdInputPointManager *
AcApDocument::inputPointManager() const;
```

The input point manager registers and deregisters input point filters, input point monitors, and input context reactors. The input point manager also enables and disables system-generated cursor graphics, so that custom cursor graphics can be drawn.

`AcEdInputPointManager` provides a function, `disableSystemCursorGraphics()`, that disables the system cursor. ObjectARX maintains a count of the calls to disable the system cursor for each document, so if your application invokes `disableSystemCursorGraphics()` multiple times, it should invoke `enableSystemCursorGraphics()` the same number of times to restore the system cursor.

WARNING! Disabling the system cursor graphics should be done sparingly, usually only when an application-defined command is prompting for user input. You must provide custom cursor graphics if you disable the system cursor graphics.

The function `disableSystemCursorGraphics()` disables the system cursor only when an input point monitor or filter provides its own cursor. This means that under normal conditions (forced entity picking is turned off), the system cursor is disabled only during point acquisition and entity selection. When forced entity picking is turned on, the system cursor is completely disabled even if there is no command active.

The input point manager also allows forced entity picking, which is the ability to track what is under the cursor during the quiescent command state. Forced entity picking can be enabled under the following conditions:

- during input point acquisition without an active object snap mode
- during single point entity picking
- during command quiescence

Finally, the input point manager contains a function, `mouseHasMoved()`, that input point filters and monitors can call to determine whether there is another digitizer event pending. If there is a digitizer event pending, the filter or monitor should return from its callback as soon as possible, without doing any further calculations, to avoid cursor lags.

Input Context Events

The `AcEdInputContextReactor` class defines callbacks that are sent to indicate the beginning and ending of various prompting operations. Input context events can be used to determine when to add and remove input point filters and monitors. They can be used independently, or with other callbacks such as `AcEditorReactor::commandWillBegin()` and `AcEditorReactor::commandEnded()`, to determine which commands are in progress, which prompt of the command is currently active, and how the user responded.

The following table describes the transitions between the different input states that can be detected using input context events:

Input context events

Type	Description
Quiescent	Entered when the beginQuiescentState() callback is made, and exited when the endQuiescentState() callback is made as a result of an AutoCAD command, AutoLISP function, or ActiveX input function being initiated. This should be the only state in the stack for a document when it is entered; it cannot be stacked on top of another state. The CMDACT system variable is zero when in this state.
Geometric, Point	Entered if the beginGetPoint() callback is made without already being in a Geometric, Nonpoint Transient, Selecting Transient, or Drag Sequence state. Whenever this state is entered, the returned point is the ultimate goal, not an intermediate value. Input point filters and monitors are called for all events in this state.
Geometric, Nonpoint	Entered from Geometric, Nonpoint Transient when the beginGetPoint() callback is made. From this state, another call to point input means stacking a new state. Any point is an intermediate value and can be overridden by a directly typed value. This state is exited when any of the endGetAngle(), endGetDistance(), endGetOrientation(), endGetCorner(), or endGetScaleFactor() callbacks are made. Input point filters and monitors are called for all events in this state.
Selecting	Entered from Selecting, Transient when the beginGetPoint() callback is made. Input point filters and monitors are called for all events in this state.

Input context events (*continued*)	
Type	**Description**
Nongeometric, Nonselecting	Entered when any of the beginGetString(), beginGetKeyword(), beginGetInteger(), beginGetColor(), or beginGetReal() callbacks are made. These contexts directly poll for input and do not perform object snap, AutoSnap, or input point filtering, even though the cursor is active when interactive digitizer tracking is performed. Forced entity picking must be enabled for input point monitors to get callbacks from this state. Input point filters are not called from this state.
Drag Sequence	Entered when the beginDragSequence() callback is made, and exited when the endDragsequence() callback is made. Nested calls to beginGetPoint(), beginGetAngle(), and beginGetDistance() are made in intermediate mode, so no state transition is made. Input point filters and monitors are called for all events in this state.
Empty Transient	The outermost input state is exited, and a new one is about to be entered.
Geometric, Nonpoint Transient	Entered when the beginGetAngle(), beginGetDistance(), beginGetOrientation(), beginGetCorner(), or beginGetScaleFactor() callbacks are made, except when already in Drag Sequence state. This means that nested prompts will return immediately and a state change is made. Entering this state implies adding an input context to the input state stack for a document. This state always transitions to Geometric, Nonpoint with the beginGetPoint() callback.
Selecting Transient	Entered when any of the beginEntsel(), beginNentsel(), or beginSSGet() callbacks are made. Will either be in the immediate mode selection state or will transition into the Selecting state with the beginGetPoint() callback.
Drag Sequence, Nested Action, Transient	Entered from Drag Sequence when the AcEditorReactor::commandWillBegin() or AcEditorReactor::LispWillStart() callbacks are made. These suspend the Drag Sequence and stack a new input state on top of it. From this state, it is possible to transition to any other input state. This stacked state will end when the balancing AcEditorReactor callback is made, and the state under the top state is Drag Sequence.

When your application is loaded, you can query the value of the system variable CMDACT to find out which commands are active. Input context events can be used to note transitions between input states.

Input Context Events Example

The following example creates a simple context reactor that responds to a variety of input context events:

```
#include <adslib.h>
#include <aced.h>
#include <dbmain.h>
#include <rxregsvc.h>
#include <acedinpt.h>
#include <acdocman.h>


// The input context reactor class.
//
class MyContextReactor : public AcEdInputContextReactor
{
public:

void
beginGetPoint(
    const AcGePoint3d* pointIn,
    const char* promptString,
    int initGetFlags,
    const char* pKeywords);

void
endGetPoint(
    Acad::PromptStatus returnStatus,
    const AcGePoint3d& pointOut,
    const char*& pKeyword);

void
beginGetOrientation(
    const AcGePoint3d* pointIn,
    const char* promptString,
    int initGetFlags,
    const char* pKeywords);

void
endGetOrientation(
    Acad::PromptStatus returnStatus,
    double& angle,
    const char*& pKeyword);

void
beginGetCorner(
    const AcGePoint3d* firstCorner,
    const char* promptString,
    int initGetFlags,
    const char* pKeywords);
```

```
void
endGetCorner(
    Acad::PromptStatus returnStatus,
    AcGePoint3d& secondCorner,
    const char*& pKeyword);

void
beginSSGet(
    const char*  pPrompt,
    int          initGetFlags,
    const char*  pKeywords,
    const char*  pSSControls,
    const AcArray<AcGePoint3d>& points,
    const resbuf* entMask);

void
endSSGet(
    Acad::PromptStatus returnStatus,
    const AcArray<AcDbObjectId>& ss);

};


void
MyContextReactor::beginGetPoint(
    const AcGePoint3d* pointIn,
    const char* promptString,
    int   initGetFlags,
    const char* pKeywords)
{
    acutPrintf("beginGetPoint: pointIn = %.2f,%.2f,%.2f\n",
        (*pointIn)[0], (*pointIn)[1], (*pointIn)[2]);

    if (NULL != promptString)
        acutPrintf("%s", promptString);
    acutPrintf("initGetFlags: %d\n", initGetFlags);
    if (NULL != pKeywords)
        acutPrintf("Keywords: %s\n", pKeywords);
}

void
MyContextReactor::endGetPoint(
    Acad::PromptStatus returnStatus,
    const AcGePoint3d& pointOut,
    const char*& pKeyword)
{
    acutPrintf("endGetPoint: %d\n", returnStatus);
    acutPrintf("%.2f,%.2f,%.2f\n", pointOut[0], pointOut[1],
        pointOut[2]);
    if (NULL != pKeyword)
        acutPrintf("Keyword: %s\n", pKeyword);
}
```

```
void
MyContextReactor::beginGetOrientation(
    const AcGePoint3d* pointIn,
    const char* promptString,
    int   initGetFlags,
    const char* pKeywords)
{
    acutPrintf("beginGetOrientation: %.2f, %.2f, %.2f\n",
        (*pointIn)[0], (*pointIn)[1], (*pointIn)[2]);
}


void
MyContextReactor::endGetOrientation(
    Acad::PromptStatus returnStatus,
    double& angle,
    const char*& pKeyword)
{
    acutPrintf("endGetOrientation: %.2f\n", angle);
}

void
MyContextReactor::beginGetCorner(
    const AcGePoint3d* firstCorner,
    const char* promptString,
    int   initGetFlags,
    const char* pKeywords)
{
    if (NULL != firstCorner)
    {
        acutPrintf(
            "beginGetCorner: %.2f, %.2f, %.2f\n",
            (*firstCorner)[0],
            (*firstCorner)[1],
            (*firstCorner)[2]);
    }
}

void
MyContextReactor::endGetCorner(
    Acad::PromptStatus returnStatus,
    AcGePoint3d& secondCorner,
    const char*& pKeyword)
{
    acutPrintf("endGetCorner\n");
}

void
MyContextReactor::beginSSGet(
    const char*  pPrompt,
    int          initGetFlags,
    const char*  pKeywords,
    const char*  pSSControls,
    const AcArray<AcGePoint3d>& points,
    const resbuf*               entMask)
```

```
{
    acutPrintf("beginSSGet:%s\n", NULL != pPrompt ? pPrompt : "");
    for (int i = 0; i < points.length(); i++)
        acutPrintf("%d: %.2f, %.2f, %.2f\n", i, points[i][X],
        points[i][Y], points[i][Z]);
}

void
MyContextReactor::endSSGet(
    Acad::PromptStatus returnStatus,
    const AcArray<AcDbObjectId>& ss)
{
    acutPrintf("endSSGet\n");
    for (int i = 0; i < ss.length(); i++)
        acutPrintf("Entity %d: <%x>\n", i, ss[i].asOldId());
}

// My context reactor object
MyContextReactor my_icr;

extern "C" __declspec(dllexport) AcRx::AppRetCode
acrxEntryPoint(
    AcRx::AppMsgCode msg,
    void *p)
{
    switch (msg)
    {
    case AcRx::kInitAppMsg:
        acrxUnlockApplication(p);
        acrxRegisterAppMDIAware(p);
        break;

    case AcRx::kLoadDwgMsg:
        // Attach a context reactor to the current document.
        //
        curDoc()->inputPointManager()->
            addInputContextReactor(&my_icr);
        break;

    case AcRx::kUnloadAppMsg:
        // Warning! This sample attaches a context reactor,
        // but it never detaches it. A real-life application
        // will need to monitor to which document it attached
        // the reactor, and will need to detach it.
        //
        break;
    }
    return AcRx::kRetOK;
}
```

Input Point Filters and Monitors

Input point filters and monitors allow you to view and modify input data as points are being acquired in AutoCAD. Input point filters and monitors are context-sensitive and should only be registered for relevant input contexts.

The following diagram shows how input point filtering and monitoring fit into the ObjectARX input processing model:

Input Point Filtering

Input point filters are invoked for all string and digitizer events when a point is being acquired. They can modify the final point position returned to the caller by AutoCAD and the ToolTip string displayed by AutoCAD.

Input point filters are processed before any input point monitors, but after all other input point computations are performed by AutoCAD. The output points are still subject to AutoCAD XYZ point filtering.

NOTE Only one input point filter can be active at a time.

Input Point Monitoring

Input point monitors are invoked for all string and digitizer events when a point is being acquired, or when forced entity picking is enabled. Input point monitors can view the final point position and can display a ToolTip string. The ToolTip string can be set by using the `appendToTooltipStr` and `additionalTooltipString` parameters when creating the input point monitor.

Input point monitors are processed after all other input point computations are performed by AutoCAD, including custom input point filtering if a filter exists. The output points are still subject to AutoCAD XYZ point filtering.

There can be any number of input point monitors active at a given time, since they only allow viewing of the input points.

Guidelines for Using Input Point Filtering and Monitoring

To ensure good performance, keep the following guidelines in mind when coding input point filters and monitors:

- Be careful about performing substantial computation from within input point filters and monitors.
- Minimize the number of callouts made from filters and monitors.
- Take advantage of available work that has already been done by AutoCAD, such as the list of all entities under the object snap cursor and all of the interim point computations.
- Be sure to free the memory for strings you pass into the `additionalTooltipString` parameter when creating an input point filter or monitor.

NOTE It is recommended that you disable your point filter in all entity selection contexts. Object snap mode and AutoSnaps are always disabled in this context.

Filter Chaining

The `AcEdInputPointFilter` class provides a member function to support chaining input point filters. Since only one filter can be active at a time, this function allows you to query the current filter, embed that filter in your custom filter, and then revoke the current filter and register your own. The following function provides the chaining functionality:

```
AcEdInputFilter *
AcEdInputPointFilter::revokeFilter(AcEdInputPointFilter *);
```

Retrying

Input point filters and monitors can be invoked multiple times for a single input point under the following conditions:

- When the TAB key is pressed after a digitizer motion event, the object snap candidate for the cursor position is cycled. As soon as the cursor motion is detected, the object snap candidate cycling index is reset.
- When a registered input point filter returns a Boolean indicating to retry for a point, the system prompts for a new event, not returning the point from the event that will be retried.

Input Point Filter and Monitor Example

The following example demonstrates the use of input point filters and monitors.

```
#include "adslib.h"
#include "aced.h'
#include "dbmain.h"
#include "acdbabb.h"
#include "adeskabb.h"
#include "rxregsvc.h"
#include "acgi.h"
#include "acdocman.h"
#include "acedinpt.h"
#include "dbapserv.h"

class IPM : public AcEdInputPointMonitor
{
public:

virtual Acad::ErrorStatus
monitorInputPoint(
    // Output. If changedTooltipStr is kTrue, newTooltipString
    // has the new ToolTip string in it.
    //
    bool&                                appendToTooltipStr,
    char*&                               additionalTooltipString,

    // Input/Output
    //
    AcGiViewportDraw*                    drawContext,

    // Input parameters:
    //
    AcApDocument*                        document,
    bool                                 pointComputed,
    int                                  history,
    const AcGePoint3d&                   lastPoint,
    const AcGePoint3d&                   rawPoint,
    const AcGePoint3d&                   grippedPoint,
    const AcGePoint3d&                   cartesianSnappedPoint,
    const AcGePoint3d&                   osnappedPoint,
    AcDb::OsnapMask                      osnapMasks,
    // const AcArray<AcDbCustomOsnapMode>& customOsnapModes
    const AcArray<AcDbObjectId>&         apertureEntities,
    const AcArray< AcDbObjectIdArray,
        AcArrayObjectCopyReallocator< AcDbObjectIdArray > >&
        nestedPickedEntities,
    int                                  gsSelectionMark,
    const AcArray<AcDbObjectId>&         keyPointEntities,
    const AcArray<AcGeCurve3d*>&         alignmentPaths,
    const AcGePoint3d&                   computedPoint,
    const char*                          tooltipString);
};
```

```
Acad::ErrorStatus
IPM::monitorInputPoint(
    // Output. If changedTooltipStr is kTrue, then newTooltipString
    // has the new ToolTip string in it.
    //
    bool&                                appendToTooltipStr,
    char*&                               additionalTooltipString,

    // Input/Output
    //
    AcGiViewportDraw*                    pDrawContext,

    // Input parameters:
    //
    AcApDocument*                        document,
    bool                                 pointComputed,
    int                                  history,
    const AcGePoint3d&                   lastPoint,
    const AcGePoint3d&                   rawPoint,
    const AcGePoint3d&                   grippedPoint,
    const AcGePoint3d&                   cartesianSnappedPoint,
    const AcGePoint3d&                   osnappedPoint,
    AcDb::OsnapMask                      osnapMasks,
    // const AcArray<AcDbCustomOsnapMode>& customOsnapModes
    const AcArray<AcDbObjectId>&         apertureEntities,
    const AcArray< AcDbObjectIdArray,
        AcArrayObjectCopyReallocator< AcDbObjectIdArray > >&
        nestedPickedEntities,
    int                                  gsSelectionMark,
    const AcArray<AcDbObjectId>&         keyPointEntities,
    const AcArray<AcGeCurve3d*>&         alignmentPaths,
    const AcGePoint3d&                   computedPoint,
    const char*                          tooltipString)
    {
        acutPrintf("\nhistory: %d\n", history);

    if (pointComputed)
    {
        acutPrintf(
            "rawPoint:                   %.2f, %.2f, %.2f\n",
            rawPoint[0],
            rawPoint[1],
            rawPoint[2]);

        if (history & Acad::eGripped)
            acutPrintf(
                "grippedPoint:           %.2f, %.2f, %.2f\n",
                grippedPoint[0],
                grippedPoint[1],
                grippedPoint[2]);
```

```
        if (history & Acad::eCartSnapped)
            acutPrintf(
                "cartesianSnappedPoint: %.2f, %.2f, %.2f\n",
                cartesianSnappedPoint[0],
                cartesianSnappedPoint[1],
                cartesianSnappedPoint[2]);

        if (history & Acad::eOsnapped)
        {
            acutPrintf(
                "osnappedPoint:          %.2f, %.2f, %.2f\n",
                osnappedPoint[0],
                osnappedPoint[1],
                osnappedPoint[2]);

#define OSMASK_CHECK(x)  if (osnapMasks & AcDb:: ## x)
    acutPrintf("%s ", #x)

            OSMASK_CHECK(kOsMaskEnd);
            OSMASK_CHECK(kOsMaskMid);
            OSMASK_CHECK(kOsMaskCen);
            OSMASK_CHECK(kOsMaskNode);
            OSMASK_CHECK(kOsMaskQuad);
            OSMASK_CHECK(kOsMaskInt);
            OSMASK_CHECK(kOsMaskIns);
            OSMASK_CHECK(kOsMaskPerp);
            OSMASK_CHECK(kOsMaskTan);
            OSMASK_CHECK(kOsMaskNear);
            OSMASK_CHECK(kOsMaskQuick);
            OSMASK_CHECK(kOsMaskApint);
            OSMASK_CHECK(kOsMaskImmediate);
            OSMASK_CHECK(kOsMaskAllowTan);
            OSMASK_CHECK(kOsMaskDisablePerp);
            OSMASK_CHECK(kOsMaskRelCartesian);
            OSMASK_CHECK(kOsMaskRelPolar);

#undef OSMASK_CHECK

            acutPrintf("\n");
        }

        acutPrintf("%d apertureEntities: ",
            apertureEntities.length());
        for (int i = 0; i < apertureEntities.length(); i++)
            acutPrintf("<%x> ", apertureEntities[i].asOldId());
        acutPrintf("\n");
    }
    else {
        acutPrintf("No point computed");
        if (history & Acad::eCyclingPt)
          acutPrintf(", but new cycling osnap: %.2f, %.2f, %.2f\n",
             osnappedPoint[0], osnappedPoint[1], osnappedPoint[2]);
        else
            acutPrintf(".\n");

    }
```

```
    if (NULL != pDrawContext)
    {
        pDrawContext->subEntityTraits().setColor(2);
        pDrawContext->geometry().circle(rawPoint, 1.0,
            AcGeVector3d::kZAxis);

    }
    else
        acutPrintf("ViewportDraw is NULL!\n");

    if (history & Acad::eNotDigitizer)
        acutPrintf("PICK!\n");

    if (NULL != tooltipString)
    {
        acutPrintf("TooltipString: %s\n", tooltipString);
        additionalTooltipString = ", anotherString!";
        appendToTooltipStr = true;
    }
    if (history & Acad::eOrtho)
    {
        acutPrintf("Ortho found at %.2f, %.2f, %.2f\n",
            computedPoint[0], computedPoint[1], computedPoint[2]);
    }
    return Acad::eOk;
}


class IPF : public AcEdInputPointFilter {
public:

Acad::ErrorStatus
processInputPoint(
    bool&                           changedPoint,
    AcGePoint3d&                    newPoint,
    bool&                           changedTooltipStr,
    char*&                          newTooltipString,
    bool&                           retry,
    AcGiViewportDraw*               pDrawContext,
    AcApDocument*                   document,
    bool                            pointComputed,
    int                             history,
    const AcGePoint3d&              lastPoint,
    const AcGePoint3d&              rawPoint,
    const AcGePoint3d&              grippedPoint,
    const AcGePoint3d&              cartesianSnappedPoint,
    const AcGePoint3d&              osnappedPoint,
    AcDb::OsnapMask                 osnapMasks,
    // const AcArray<AcDbCustomOsnapMode>& customOsnapModes
    const AcArray<AcDbObjectId>&    pickedEntities,
    const AcArray< AcDbObjectIdArray,
        AcArrayObjectCopyReallocator< AcDbObjectIdArray > >&
        nestedPickedEntities,
    // Of 0th element in pickedEntities.
```

```
    int       gsSelectionMark,
    // AutoSnap Info:
    const AcArray<AcDbObjectId>&     keyPointEntities,
    const AcArray<AcGeCurve3d*>&     alignmentPaths,

    const AcGePoint3d&               computedPoint,
    const char*                      tooltipString);
};


Acad::ErrorStatus
IPF::processInputPoint(
    bool&                            changedPoint,
    AcGePoint3d&                     newPoint,
    bool&                            changedTooltipStr,
    char*&                           newTooltipString,
    bool&                            retry,
    AcGiViewportDraw*                pDrawContext,
    AcApDocument*                    document,
    bool                             pointComputed,
    int                              history,
    const AcGePoint3d&               lastPoint,
    const AcGePoint3d&               rawPoint,
    const AcGePoint3d&               grippedPoint,
    const AcGePoint3d&               cartesianSnappedPoint,
    const AcGePoint3d&               osnappedPoint,
    // const AcArray<AcDbCustomOsnapMode>& customOsnapModes
    const AcArray<AcDbObjectId>&     pickedEntities,
    const AcArray< AcDbObjectIdArray,
       AcArrayObjectCopyReallocator< AcDbObjectIdArray > >&
       nestedPickedEntities,
    // Of 0th element in pickedEntities.
    int                              gsSelectionMark,
    // AutoSnap Info:
    const AcArray<AcDbObjectId>&     keyPointEntities,
    const AcArray<AcGeCurve3d*>&     alignmentPaths,
    const AcGePoint3d&               computedPoint,
    const char*                      tooltipString)
{
    // Change the computed point to an offset of (0.2, 0.2, 0.2)
    // if the current computed point is an object snap point.
    //
    if (pointComputed && history & Acad::eOsnapped)
    {
        changedPoint = true;
        newPoint = osnappedPoint + AcGeVector3d(0.2,0.2,0.0);
        pDrawContext->geometry().circle(newPoint, 0.1,
            AcGeVector3d::kZAxis);
    }

    return Acad::eOk;
}
```

```
// Input point monitor
IPM my_ipm;

// Input point filter
IPF my_ipf;


// Installs an input point monitor.
//
void testipm()
{
    curDoc()->inputPointManager()->addPointMonitor(&my_ipm);
}


// Installs an input point filter.
//
void testipf()
{
    curDoc()->inputPointManager()->registerPointFilter(&my_ipf);
}


// Turns on forced entity picking.
//
void testfp()
{
    curDoc()->inputPointManager()->turnOnForcedPick();
}


// Disables the system cursor graphics.
//
void testcursor()
{
    curDoc()->inputPointManager()->disableSystemCursorGraphics();
}
```

```
extern "C" __declspec(dllexport)
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void *p)
{
    switch (msg)
    {
    case AcRx::kInitAppMsg:
        acrxRegisterAppMDIAware(p);
        acrxUnlockApplication(p);
        acedRegCmds->addCommand("mkr", "testipm", "ipm",
            ACRX_CMD_TRANSPARENT, testipm);
        acedRegCmds->addCommand("mkr", "testipf", "ipf",
            ACRX_CMD_TRANSPARENT, testipf);
        acedRegCmds->addCommand("mkr", "testfp", "fp",
            ACRX_CMD_TRANSPARENT, testfp);
        acedRegCmds->addCommand("mkr", "testcursor", "cursor",
            ACRX_CMD_TRANSPARENT, testcursor);
        break;

    case AcRx::kUnloadAppMsg:
        acedRegCmds->removeGroup("mkr");
        break;
    }
    return AcRx::kRetOK;
}
```

Application Configuration

In This Chapter

22

- Profile Manager

This chapter discusses configuring your application for the end user.

Profile Manager

The Profile Manager allows you to perform all the operations provided in the Profiles tab of the Options Dialog. This API consists of two classes and several methods used to manipulate user profiles easily.

AcApProfileManager Class

There is only one Profile Manager object per AutoCAD session. ObjectARX provides a global function to get access to this Profile Manager object, called `acProfileManagerPtr()`. This function returns a pointer to an `AcApProfileManager` object, upon which the methods can then be called.

The `AcApProfileManager` class provides a container for the operations provided by the Profiles tab. There is no method provided to obtain the current profile name, since this is stored in the system variable called CPROFILE and can be obtained using a call to the `acutGetVar()`.

AcApProfileManager class provides the following capabilities

Capabilities	Associated Method
Retrieve the registry path for a specified profile	AcApProfileManager::ProfileRegistryKey
Retrieve the profile names that currently exist for a user's configuration	AcApProfileManager::ProfileListNames
Export a given profile to an AutoCAD registry profile file (*.arg*) in REGEDIT4 format	AcApProfileManager::ProfileExport
Import an AutoCAD registry profile file (*.arg*) into a given profile	AcApProfileManager::ProfileImport
Delete a specified profile from the registry	AcApProfileManager::ProfileDelete
Reset a specified profile to AutoCAD default settings	AcApProfileManager::ProfileReset
Make a specified profile active or current for the AutoCAD session	AcApProfileManager::ProfileSetCurrent

AcApProfileManager class provides the following capabilities (*continued*)	
Capabilities	**Associated Method**
Copy an existing profile to a new profile	AcApProfileManager::ProfileCopy
Rename an existing profile	AcApProfileManager::ProfileRename
Add a new profile reactor object	AcApProfileManager::addReactor
Remove an existing profile reactor object	AcApProfileManager::removeReactor

AcApProfileManagerReactor Class

The `AcApProfileManagerReactor` class provides a container for different event notifications based on user profile changes.

AcApProfileManagerReactor class notifications	
Notification	**Associated Method**
The current profile is about to be changed	AcApProfileManagerReactor::currentProfileWillChange
The current profile has been changed	AcApProfileManagerReactor::currentProfileChanged
The current profile is about to be reset	AcApProfileManagerReactor::currentProfileWillBeReset
The current profile has been reset	AcApProfileManagerReactor::currentProfileReset
A non-current profile is about to be reset	AcApProfileManagerReactor::profileWillReset
A non-current profile has been reset	AcApProfileManagerReactor::profileReset

Profile Manager Sample

The following sample demonstrates the use of the `AcApProfileManager` and `AcApProfileManagerReactor` classes:

```
// Define a class derived from AcApProfileManagerReactor to manage
// the notifications.
//
class AsdkProfileManagerReactor : public AcApProfileManagerReactor
{
public:
    void currentProfileWillChange(const char *newProfile);
    void currentProfileChanged(const char *newProfile);
    void currentProfileWillBeReset(const char *curProfile);
    void currentProfileReset(const char *curProfile);
    void profileWillReset(const char *profName);
    void profileReset(const char *proName);
};

// Define the notification functions.
//
void
AsdkProfileManagerReactor::
currentProfileWillChange(const char *newProfile)
{
    acutPrintf("\nCurrent profile will change: %s", newProfile);
}
void
AsdkProfileManagerReactor::
currentProfileChanged(const char *newProfile)
{
    acutPrintf("\nCurrent profile changed: %s", newProfile);
}

void
AsdkProfileManagerReactor::
currentProfileWillBeReset(const char *curProfile)
{
    acutPrintf("\nCurrent profile will be reset: %s", curProfile);
}

void
AsdkProfileManagerReactor::
currentProfileReset(const char *curProfile)
{
   acutPrintf("\nCurrent profile has been reset: %s", curProfile);
}

void
AsdkProfileManagerReactor::
profileWillReset(const char *profName)
{
   acutPrintf("\nNon-current profile will be reset: %s", profName);
}
```

```
void
AsdkProfileManagerReactor::
profileReset(const char *profName)
{
   acutPrintf("\nNon-current profile has been reset:%s", profName);
}

void
aFunction()
{
    acutPrintf("This is AsdkProfileSample Test Application...\n");

    // Attach the reactor for the duration of this command. Normally
    // this would be added upon application initialization.
    //
    AsdkProfileManagerReactor *pProfileRector =
        new AsdkProfileManagerReactor();

    acProfileManagerPtr()->addReactor(pProfileRector);

    // Obtain the path for the registry keys and print it out.
    //
    char *pstrKey;
    acProfileManagerPtr()->ProfileRegistryKey(pstrKey, NULL);
    if (pstrKey != NULL) {
        acutPrintf("\nThe profiles registry key is: %s", pstrKey);
        acutDelString(pstrKey);
    }

    // Get the list of all profiles in the user's configuration
    // and print them out.
    //
    AcApProfileNameArray arrNameList;
    int nProfiles =
        acProfileManagerPtr()->ProfileListNames(arrNameList);
    acutPrintf("\nNumber of profiles currently "
        "in the user profile list is: %d", nProfiles);
    for (int i = 0; i < nProfiles; i++)
        acutPrintf("\nProfile name is: %s", arrNameList[i]);

    // Copy the unnamed profile to the AsdkTestProfile.
    //
    acProfileManagerPtr()->ProfileCopy(
        "AsdkTestProfile",
        "<<Unnamed Profile>>",
        "This is a test");

    // Reset the newly copied profile to AutoCAD defaults.
    //
    acProfileManagerPtr()->ProfileReset("AsdkTestProfile");
```

```
    // Make this new profile current.
    //
    acProfileManagerPtr()->ProfileSetCurrent("AsdkTestProfile");
    // Change a value in the profile. For this example, make the
    // cursor big.
    //
    struct resbuf rbCursorSize;
    rbCursorSize.restype = RTSHORT;
    rbCursorSize.resval.rint = 100;
    acedSetVar("CURSORSIZE", &rbCursorSize);

    // Rename the profile to a new name.
    //
    acProfileManagerPtr()->ProfileRename(
        "AsdkTestProfile2",
        "AsdkTestProfile",
        "This is another test");

    // Export the profile.
    //
    acProfileManagerPtr()->ProfileExport(
        "AsdkTestProfile2",
        "./AsdkTestProfile2.arg");
    // Import the profile.
    //
    acProfileManagerPtr()->ProfileImport(
        "AsdkTestProfile3",
        "./AsdkTestProfile2.arg",
        "This is a copy of AsdkTestProfile2"
        "by Exporting/Importing",
        Adesk::kTrue);

    // Remove the reactor.
    //
    acProfileManagerPtr()->removeReactor(pProfileRector);
}
```

Part V

Interacting with Other Environments

COM, ActiveX Automation, and the Object Property Manager

23

In This Chapter

- Overview
- Using AutoCAD COM Objects from ObjectARX and Other Environments
- AutoCAD ActiveX Automation Implementation
- Interacting with AutoCAD
- Document Locking
- Creating a Registry File
- Exposing Automation Functionality
- Object Property Manager API
- Static OPM COM Interfaces
- Implementing Static OPM Interfaces
- Dynamic Properties and OPM

The Microsoft Component Object Model (COM) allows applications on Windows-based platforms to communicate and exchange data with each other.

You can access COM interfaces provided by other ObjectARX applications or by any COM-enabled application running on Windows.

Additionally, by using COM with ObjectARX, you can augment AutoCAD's ActiveX Automation model. The elements, objects, or entities you expose will then be available to other programming environments such as Visual Basic for Applications (VBA) and to AutoCAD features such as the Object Property Manager (OPM).

Overview

Microsoft's Component Object Model (COM) was originally designed to support Object Linking and Embedding (OLE); it also became the basis of ActiveX Automation. As the emergent standard for Windows component development, COM has relevance beyond OLE and ActiveX. Component architecture separates interface from implementation, allowing applications to consist of dynamically linked components rather than a single binary executable. Developers can write programs that take advantage of existing COM components, or they can use COM to create their own components.

ObjectARX applications can be designed as COM clients. For instance, an ObjectARX application that needs to communicate with another program could implement COM access to that program. Depending on the COM interfaces that the other application provides, the ObjectARX application could then exchange information with that application or even drive it.

An ObjectARX application can also act as an Automation server. You can write COM wrappers to expose additional elements or custom ObjectARX objects. New APIs, templates, classes, and support for the Microsoft Active Template Library (ATL) make it easier than ever to add to the AutoCAD ActiveX Automation model.

Using AutoCAD COM Objects from ObjectARX and Other Environments

AutoCAD provides COM wrappers for most of the ObjectARX environment. These objects are ActiveX Automation compliant, which supports Visual Basic, Java, C++, and any other Windows environment that can access ActiveX objects. As such, AutoCAD has provided some APIs that are available ONLY through the COM interface mechanism. See the *ActiveX and VBA Developer's Guide* for documentation relating to the ActiveX Object Model.

The following AutoCAD features provide aspects of their APIs as COM interfaces:

- Hardcopy
- Menus
- Options

Accessing COM Interfaces from ObjectARX

Some APIs are only available as ActiveX, so in order to access them from ObjectARX, you need to use COM from C++. This section explains the two methods you can use to do so. The first is to use MFC and the Visual C++ ClassWizard to read the AutoCAD type library. This type library (*acad.tlb*) contains the ActiveX Object model. The second method requires a bit more work, but doesn't require the use of MFC.

Using MFC and ClassWizard to Access AutoCAD ActiveX Automation

This method uses MFC and the Visual C++ ClassWizard to read the AutoCAD type library (*acad.tlb*).

To call the ActiveX Automation interfaces using MFC and the ClassWizard Type Library Importation system

1. The sample program will use the COM ActiveX Automation interfaces of AutoCAD to create a circle in model space. Start Visual C++ and create a new MFC AppWizard(dll) project called **AsdkComMfcDocSamp**.
2. Choose Regular DLL using shared MFC DLL.

NOTE You can actually choose any of the options, but different settings and code will be required depending on your choice. This example will use the Regular DLL using shared MFC DLL. See chapter 8, "MFC Topics," for more information on options to choose for different tasks.

---

3. Select Finish and then OK to create the project.
4. Add the appropriate values to the project settings to make the project build as an ObjectARX program. This program needs to link with the following libraries:

```
acad.lib
rxapi.lib
acedapi.lib
```

5. Add the appropriate lines to the DEF file EXPORTS section:

```
acrxEntryPoint
_SetacrxPtp
acrxGetApiVersion
```

6 Open the *AsdkComMfcDocSamp.cpp* source file and add the following code to make the program ObjectARX compatible. Notice the macro call in the `acrxEntryPoint()` function for `AFX_MANAGE_STATE(AfxGetStaticModuleState())`:

```
static void initApp()
{
    acedRegCmds->addCommand(
        "ASDK_MFC_COM",
        "AsdkMfcComCircle",
        "MfcComCircle",
        ACRX_CMD_MODAL,
        addCircleThroughMfcCom);
}

static void unloadApp()
{
    acedRegCmds->removeGroup("ASDK_MFC_COM");
}

extern "C" AcRx::AppRetCode acrxEntryPoint
    (AcRx::AppMsgCode msg, void* appId)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    switch(msg)
    {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
        break;
    default:
        break;
    }
    return AcRx::kRetOK;
}
```

7 The next step is to decide which interfaces are necessary to get the circle into model space. In this case, the `IAcadApplication`, `IAcadDocument`, and `IAcadModelSpace` interfaces are required. To get the definitions of these interfaces, use the AutoCAD type library (*acad.tlb*). First choose ClassWizard from the View menu. Then choose Add Class and pick From a Type Library. In the Import from Type Library dialog, choose the *acad.tlb* file from the root AutoCAD directory and choose Open. From the Confirm Classes dialog, multiselect the `IAcadApplication`, `IAcadDocument`, and `IAcadModelSpace` interface classes. The header and implementation file will default to *acad.h* and *acad.cpp*, respectively. Click OK and the ClassWizard will import these interface classes from the type library.

8 Open the *acad.cpp* and *acad.h* files and explore the classes and methods that were imported.

NOTE All the ActiveX Automation interfaces are documented in the ActiveX and VBA Reference.

---

9 Open the *AsdkComMfcDocSamp.cpp* file and add the following function to the file:

```
void addCircleThroughMfcCom()
{
}
```

10 Add the declarations for the three interface classes:

```
IAcadApplication IApp;
IAcadDocument IDoc;
IAcadModelSpace IMSpace;
```

11 Use the `acedGetAcadWinApp` to obtain the `CWinApp` MFC object for AutoCAD and call the `GetIDispatch` method.

```
IDispatch *pDisp = acedGetAcadWinApp()->
    GetIDispatch(TRUE);
```

12 Once you have the `IDispatch` object, attach it to the locally defined `IAcadApplication` object and make sure that AutoCAD is visible:

```
IApp.AttachDispatch(pDisp);
IApp.SetVisible(true);
```

13 Obtain the active document dispatch and attach it to the locally defined `IAcadDocument` object:

```
pDisp = IApp.GetActiveDocument();
IDoc.AttachDispatch(pDisp);
```

14 Query the active document for model space.

```
pDisp = IDoc.GetModelSpace();
IMSpace.AttachDispatch(pDisp);
```

15 A circle requires a center point and radius. To make this efficient and transparent to different programming languages, the COM interface uses the `VARIANT` type. A point is stored in a `VARIANT` as a `SAFEARRAY`. The following code sets up a `SAFEARRAY` and stores it in a `VARIANT`:

```
SAFEARRAYBOUND rgsaBound;
rgsaBound.lLbound = 0L;
rgsaBound.cElements = 3;
SAFEARRAY* pStartPoint = NULL;
pStartPoint = SafeArrayCreate(VT_R8, 1, &rgsaBound);
```

```
// X value.
//
long i = 0;
double value = 4.0;
SafeArrayPutElement(pStartPoint, &i, &value);

// Y value.
//
i++;
value = 2.0;
SafeArrayPutElement(pStartPoint, &i, &value);

// Z value.
//
i++;
value = 0.0;
SafeArrayPutElement(pStartPoint, &i, &value);

VARIANT pt1;
VariantInit(&pt1);
V_VT(&pt1) = VT_ARRAY | VT_R8;
V_ARRAY(&pt1) = pStartPoint;
```

16 Call the `AddCircle` method from the `IAcadModelSpace` object:

```
IMSpace.AddCircle(pt1, 2.0);
```

The entire function should now look like

```
void addCircleThroughMfcCom()
{
    IAcadApplication IApp;

    IAcadDocument IDoc;
    IAcadModelSpace IMSpace;
    IDispatch *pDisp = acedGetAcadWinApp()->
        GetIDispatch(FALSE);
    IApp.AttachDispatch(pDisp);
    IApp.SetVisible(true);
    pDisp = IApp.GetActiveDocument();
    IDoc.AttachDispatch(pDisp);
    pDisp = IDoc.GetModelSpace();
    IMSpace.AttachDispatch(pDisp);

    SAFEARRAYBOUND rgsaBound;
    rgsaBound.lLbound = 0L;
    rgsaBound.cElements = 3;
    SAFEARRAY* pStartPoint = NULL;
    pStartPoint = SafeArrayCreate(VT_R8, 1, &rgsaBound);
    // X value
    long i = 0;
    double value = 4.0;
    SafeArrayPutElement(pStartPoint, &i, &value);
    // Y value
    i++;
    value = 2.0;
    SafeArrayPutElement(pStartPoint, &i, &value);
```

```
    // Z value
    i++;
    value = 0.0;
    SafeArrayPutElement(pStartPoint, &i, &value);

    VARIANT pt1;
    VariantInit(&pt1);
    V_VT(&pt1) = VT_ARRAY | VT_R8;
    V_ARRAY(&pt1) = pStartPoint;
    IMSpace.AddCircle(pt1, 2.0);
}
```

Using COM to Access AutoCAD ActiveX Automation

This method requires more coding but doesn't rely on MFC.

To call the ActiveX Automation interfaces without MFC

1. The sample program will use the COM ActiveX Automation interfaces to add a new pop-up menu to the AutoCAD menu bar. Start Visual C++ and create a new Win32 Dynamic-Link Library project called **AsdkComDocSamp**.
2. Add the appropriate values to the project settings to make the project build as an ObjectARX program. This program needs to link with the following libraries:

```
acad.lib
rxapi.lib
acrx15.lib
acutil15.lib
acedapi.lib
```

3. Add a new definition file to the project called *AsdkComDocSamp.def* and add the following lines:

```
DESCRIPTION 'Autodesk AsdkCom ARX test application'
LIBRARY     AsdkComDocSamp
EXPORTS     acrxEntryPoint
            _SetacrxPtp
            acrxGetApiVersion
```

4. Add a new source file to the project called *AsdkComDocSamp.cpp* and add the following code to make the program ObjectARX compatible:

```
#include <rxregsvc.h>
#include <aced.h>
#include <adslib.h>

// Used to add/remove the menu with the same command.
//
static bool bIsMenuLoaded = false;
```

```
void
addMenuThroughCom()
{
}
static void initApp()
{
    acedRegCmds->addCommand(
        "ASDK_PLAIN_COM",
        "AsdkComMenu",
        "ComMenu",
        ACRX_CMD_MODAL,
        addMenuThroughCom);
}

static void unloadApp()
{
    acedRegCmds->removeGroup("ASDK_PLAIN_COM");
}

extern "C" AcRx::AppRetCode acrxEntryPoint
    (AcRx::AppMsgCode msg, void* appId)
{
    switch( msg )
    {
        case AcRx::kInitAppMsg:
            acrxDynamicLinker->unlockApplication(appId);
            acrxDynamicLinker->registerAppMDIAware(appId);
            initApp();
            break;
        case AcRx::kUnloadAppMsg:
            unloadApp();
            break;
        default:
            break;
    }
    return AcRx::kRetOK;
}
```

5 Import the AutoCAD type library to pick up the definitions for the COM objects. Add the following line to the top of the *AsdkComDocSamp.cpp* file. Make sure to use the path for the AutoCAD installed on your system:

```
#import "c:\\acad\\acad.tlb" no_implementation \
    raw_interfaces_only named_guids
```

6 Decide which interfaces you will need to access. Since this example uses the AutoCAD menu bar, it requires most of the menu objects. These are declared in the `addMenuThroughCom` function as follows:

```
AutoCAD::IAcadApplication *pAcad;
AutoCAD::IAcadMenuBar *pMenuBar;
AutoCAD::IAcadMenuGroups *pMenuGroups;
AutoCAD::IAcadMenuGroup *pMenuGroup;
AutoCAD::IAcadPopupMenus *pPopUpMenus;
AutoCAD::IAcadPopupMenu *pPopUpMenu;
AutoCAD::IAcadPopupMenuItem *pPopUpMenuItem;
```

7 The more direct COM approach to access the Automation interfaces uses `QueryInterface`. The following code returns the `IUnknown` for AutoCAD:

```
HRESULT hr = NOERROR;
CLSID clsid;
LPUNKNOWN pUnk = NULL;
LPDISPATCH pAcadDisp = NULL;
hr = ::CLSIDFromProgID(L"AutoCAD.Application", &clsid);
if (SUCCEEDED(hr))
{
    if(::GetActiveObject(clsid, NULL, &pUnk) == S_OK)
    {
        if (pUnk->QueryInterface(IID_IDispatch,
            (LPVOID*) &pAcadDisp) != S_OK)
            return;
            pUnk->Release();
    }
}
```

8 Use `IUnknown` to get the AutoCAD application object. Also, make sure AutoCAD is visible and get the `IAcadMenuBar` and `IAcadMenuGroups` objects. This is shown in the following code:

```
    if (SUCCEEDED(pAcadDisp->QueryInterface
        (AutoCAD::IID_IAcadApplication,(void**)&pAcad))) {
        pAcad->put_Visible(true);
    }else {
        acutPrintf("\nQueryInterface trouble.");
        return;
    }
```

9 With the AutoCAD application, get the menu bar and menu groups collections. Determine how many menus are current on the menu bar:

```
pAcad->get_MenuBar(&pMenuBar);
pAcad->get_MenuGroups(&pMenuGroups);
pAcad->Release();
long numberOfMenus;
pMenuBar->get_Count(&numberOfMenus);
pMenuBar->Release();
```

10 Get the first menu from the menu groups collection. This will normally be `ACAD`, but could be something else. Then get the pop-up menus collection from the first menu group:

```
VARIANT index;
VariantInit(&index);
V_VT(&index) = VT_I4;
V_I4(&index) = 0;
pMenuGroups->Item(index, &pMenuGroup);
pMenuGroups->Release();
pMenuGroup->get_Menus(&pPopUpMenus);
pMenuGroup->Release();
```

11 Depending on whether the menu is already created, either construct a new pop-up menu or remove the previously created one. The following code completes the example:

```
WCHAR wstrMenuName[256];
MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED,
    "AsdkComAccess", -1, wstrMenuName, 256);
if (!bIsMenuLoaded) {
    pPopUpMenus->Add(wstrMenuName, &pPopUpMenu);
    if (pPopUpMenu != NULL) {
        pPopUpMenu->put_Name(wstrMenuName);
        WCHAR wstrMenuItemName[256];
        MultiByteToWideChar(CP_ACP, 0,"&Add A ComCircle",
            -1, wstrMenuItemName, 256);
        WCHAR wstrMenuItemMacro[256];
        MultiByteToWideChar(CP_ACP, 0, "AsdkComCircle ",
            -1, wstrMenuItemMacro, 256);
        VariantInit(&index);
        V_VT(&index) = VT_I4;
        V_I4(&index) = 0;
        pPopUpMenu->AddMenuItem(index, wstrMenuItemName,
            wstrMenuItemMacro, &pPopUpMenuItem);
        VariantInit(&index);
        V_VT(&index) = VT_I4;
        V_I4(&index) = 1;
        pPopUpMenu->AddSeparator(index,
            &pPopUpMenuItem);
        MultiByteToWideChar(CP_ACP, 0,
            "Auto&LISP Example", -1,
            wstrMenuItemName, 256);
        MultiByteToWideChar(CP_ACP, 0,
            "(prin1 \"Hello\") ", -1,
            wstrMenuItemMacro, 256);
        VariantInit(&index);
        V_VT(&index) = VT_I4;
        V_I4(&index) = 2;
        pPopUpMenu->AddMenuItem(index, wstrMenuItemName,
            wstrMenuItemMacro, &pPopUpMenuItem);
        VariantInit(&index);
        V_VT(&index) = VT_I4;
        V_I4(&index) = numberOfMenus - 2;;
        pPopUpMenu->InsertInMenuBar(index);
        pPopUpMenu->Release();
        pPopUpMenuItem->Release();
        bIsMenuLoaded = true;
        }else {
        acutPrintf("\nMenu not created.");
    }
    }else {
    VariantInit(&index);
    V_VT(&index) = VT_BSTR;
    V_BSTR(&index) = wstrMenuName;
    pPopUpMenus->RemoveMenuFromMenuBar(index);

    bIsMenuLoaded = false;
}
pPopUpMenus->Release();
```

The entire function should now look like

```
void
addMenuThroughCom()
{
    AutoCAD::IAcadApplication *pAcad;
    AutoCAD::IAcadMenuBar *pMenuBar;
    AutoCAD::IAcadMenuGroups *pMenuGroups;
    AutoCAD::IAcadMenuGroup *pMenuGroup;
    AutoCAD::IAcadPopupMenus *pPopUpMenus;
    AutoCAD::IAcadPopupMenu *pPopUpMenu;
    AutoCAD::IAcadPopupMenuItem *pPopUpMenuItem;
    HRESULT hr = NOERROR;
    CLSID clsid;
    LPUNKNOWN pUnk = NULL;
    LPDISPATCH pAcadDisp = NULL;

    hr = ::CLSIDFromProgID(L"AutoCAD.Application",
        &clsid);
    if (SUCCEEDED(hr))
    {
        if(::GetActiveObject(clsid, NULL, &pUnk) == S_OK)
        {
            if (pUnk->QueryInterface(IID_IDispatch,
                    (LPVOID*) &pAcadDisp) != S_OK)
                return;
            pUnk->Release();
        }
    }

    if (SUCCEEDED(pAcadDisp->QueryInterface
        (AutoCAD::IID_IAcadApplication,(void**)&pAcad))) {
        pAcad->put_Visible(true);
    }else {
        acutPrintf("\nQueryInterface trouble.");
        return;
    }
    pAcad->get_MenuBar(&pMenuBar);
    pAcad->get_MenuGroups(&pMenuGroups);
    pAcad->Release();
    long numberOfMenus;
    pMenuBar->get_Count(&numberOfMenus);
    pMenuBar->Release();
    VARIANT index;
    VariantInit(&index);
    V_VT(&index) = VT_I4;
    V_I4(&index) = 0;
    pMenuGroups->Item(index, &pMenuGroup);
    pMenuGroups->Release();

    pMenuGroup->get_Menus(&pPopUpMenus);
    pMenuGroup->Release();
    WCHAR wstrMenuName[256];
    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED,
        "AsdkComAccess", -1, wstrMenuName, 256);
```

```
    if (!bIsMenuLoaded) {
        pPopUpMenus->Add(wstrMenuName, &pPopUpMenu);
        if (pPopUpMenu != NULL) {
            pPopUpMenu->put_Name(wstrMenuName);
            WCHAR wstrMenuItemName[256];
            MultiByteToWideChar(CP_ACP, 0,"&Add A ComCircle",
                -1, wstrMenuItemName, 256);
            WCHAR wstrMenuItemMacro[256];
            MultiByteToWideChar(CP_ACP, 0, "AsdkComCircle ",
                -1, wstrMenuItemMacro, 256);
            VariantInit(&index);
            V_VT(&index) = VT_I4;
            V_I4(&index) = 0;
            pPopUpMenu->AddMenuItem(index, wstrMenuItemName,
                wstrMenuItemMacro, &pPopUpMenuItem);
            VariantInit(&index);
            V_VT(&index) = VT_I4;
            V_I4(&index) = 1;
            pPopUpMenu->AddSeparator(index,
                &pPopUpMenuItem);
            MultiByteToWideChar(CP_ACP, 0,
                "Auto&LISP Example", -1,
                wstrMenuItemName, 256);
            MultiByteToWideChar(CP_ACP, 0,
                "(prin1 \"Hello\") ", -1,
                wstrMenuItemMacro, 256);
            VariantInit(&index);
            V_VT(&index) = VT_I4;
            V_I4(&index) = 2;
            pPopUpMenu->AddMenuItem(index, wstrMenuItemName,
                wstrMenuItemMacro, &pPopUpMenuItem);
            VariantInit(&index);
            V_VT(&index) = VT_I4;
            V_I4(&index) = numberOfMenus - 2;;
            pPopUpMenu->InsertInMenuBar(index);
            pPopUpMenu->Release();
            pPopUpMenuItem->Release();

            bIsMenuLoaded = true;
        } else {
            acutPrintf("\nMenu not created.");
        }
    } else {
        VariantInit(&index);
        V_VT(&index) = VT_BSTR;
        V_BSTR(&index) = wstrMenuName;
        pPopUpMenus->RemoveMenuFromMenuBar(index);

        bIsMenuLoaded = false;
    }
    pPopUpMenus->Release();

}
```

Both of these examples can be found in the ObjectARX SDK. They are located in the *docsamps\COM* directory. Each sample contains code for adding a cir-

cle and a menu using either Win32 API or MFC programming techniques. Since these methods are accessing AutoCAD through COM interfaces, these programming techniques can be used from other C++ programs (not just ObjectARX). Also, other languages such as Java and Visual Basic can be used.

AutoCAD ActiveX Automation Implementation

AutoCAD's ActiveX Automation model is becoming increasingly visible across all of the programming environments it supports. Understanding AutoCAD's Automation capability will allow you to build applications that take full advantage of VBA and features like OPM and AutoCAD DesignCenter.

The Relationship between AcDbObjects and Automation Objects

AutoCAD implements its ActiveX Automation object model by creating a link between a database-resident object (`AcDbObject`) and a COM object that represents it. This link is composed of two one-directional pointers. The first pointer is the `IUnknown` of the COM object, which is stored using a transient reactor on the `AcDbObject`. The second pointer is the `AcDbObjectId` of the database-resident object, which is stored as a member variable on the COM object.

This link allows you to retrieve the existing `IUnknown` pointer of the COM object given an `AcDbObject` pointer, as shown in the following code:

```
AcAxOleLinkManager* pOleLinkManager = AcAxGetOleLinkManager();
// pObject is an AcDbObject*
//
IUnknown* pUnk = pOleLinkManager->GetIUnknown(pObject);

// NOTE: AcAxOleLinkManager::GetIUnknown() does not AddRef()
// the IUnknown pointer.
```

Conversely, you can retrieve the `AcDbObjectId` of the database-resident object being represented by a COM object given an `IUnknown` pointer, as shown in the following code:

```
IAcadBaseObject* pAcadBaseObject = NULL;
// pUnk is the IUnknown* of a COM object representing
// some object in the database
//
HRESULT hr = pUnk->QueryInterface(IID_IAcadBaseObject, (LPVOID*)
&pAcadBaseObject);
AcDbObjectId objId;
if(SUCCEEDED(hr))
{
    pAcadBaseObject->GetObjectId(&objId);
}
```

IAcadBaseObject

`IAcadBaseObject` is the interface used to manage the link from a COM object to a database-resident object. It is the COM object's responsibility to reset the link from the `AcDbObject` to the COM object when the COM object is being destroyed. This is done using the `AcAxOleLinkManager` class discussed below, usually in the destructor of the COM class:

```
interface DECLSPEC_UUID("5F3C54C0-49E1-11cf-93D5-0800099EB3B7")
IAcadBaseObject : public IUnknown
{
    // IUnknown methods
    //
    STDMETHOD(QueryInterface)(THIS_ REFIID riid, LPVOID FAR*
        ppvObj) PURE;
    STDMETHOD_(ULONG, AddRef)(THIS) PURE;
    STDMETHOD_(ULONG, Release)(THIS) PURE;

    // IAcadBaseObject methods
    //
    STDMETHOD(SetObjectId)(THIS_ AcDbObjectId& objId,
        AcDbObjectId ownerId = AcDbObjectId::kNull,
        TCHAR* keyName = NULL) PURE;
    STDMETHOD(GetObjectId)(THIS_ AcDbObjectId* objId) PURE;
    STDMETHOD(Clone)(THIS_ AcDbObjectId ownerId,
        LPUNKNOWN* pUnkClone) PURE;
    STDMETHOD(GetClassID)(THIS_ CLSID& clsid) PURE;
    STDMETHOD(NullObjectId)(THIS) PURE;
    STDMETHOD(OnModified)(THIS) PURE;
};
```

SetObjectId()

This method is used to identify which database-resident object the COM object represents. If the `objId` argument is equal to `AcDbObjectId::kNull`, the COM object is being instructed to create a new `AcDbObject`-derived object and append it to the database. The `ownerId` and `keyName` arguments are only specified in this situation.

GetObjectId()
This method is used to retrieve the `AcDbObjectId` of the database-resident object being represented.

Clone()
This method is reserved for future use.

GetClassID()
This method returns the `CLSID` of the COM object.

NullObjectId()
This method is used to tell the COM object that it is no longer representing a database-resident object.

OnModified()
This method is used to tell the COM object that the `AcDbObject` it represents has been modified. The COM object is then responsible for firing notification to all its clients through established connection points.

AcAxOleLinkManager

`AcAxOleLinkManager` is used to manage the link from the database-resident object to its COM object. This is done by attaching a transient reactor to the `AcDbObject`. The transient reactor has one variable containing the `IUnknown` of the COM object. This transient reactor is also used to call `IAcadBaseObject::OnModified()` when the `AcDbObject` is modified.

To get a pointer to the OLE link manager, use `AcAxGetOleLinkManager()`. The `AcAxOleLinkManager` class is described below:

```
// AcAxOleLinkManager is used to maintain the link between ARX
// objects and their respective COM wrapper.
//
class AcAxOleLinkManager
{
public:

    // Given a pointer to a database-resident object, return
    // the IUnknown of the COM wrapper. NULL is returned if
    // no wrapper is found.
    //
    virtual IUnknown* GetIUnknown(AcDbObject* pObject) = 0;

    // Set the link between a database-resident object and a
    // COM wrapper. If the IUnknown is NULL, then the link
    // is removed.
    //
    virtual Adesk::Boolean SetIUnknown(AcDbObject* pObject,
        IUnknown* pUnknown) = 0;
```

```
    // Given a pointer to a database object, return
    // the IUnknown of the COM wrapper. NULL is returned if
    // no wrapper is found.
    //
    virtual IUnknown* GetIUnknown(AcDbDatabase* pDatabase) = 0;

    // Set the link between a database object and a COM wrapper.
    // If the IUnknown is NULL, then the link is removed.
    //
    virtual Adesk::Boolean SetIUnknown(AcDbDatabase* pDatabase,
        IUnknown* pUnknown) = 0;

    // Given a pointer to a database object, return the
    // IDispatch of the document object. NULL is returned if
    // the database does not belong to a particular document.
    //
    virtual IDispatch* GetDocIDispatch(AcDbDatabase* pDatabase)= 0;

    // Set the link between a database object and the IDispatch
    // of the document it belongs to. If the IDispatch is NULL, then
    // the link is removed.
    //
   virtual Adesk::Boolean SetDocIDispatch(AcDbDatabase* pDatabase,
        IDispatch* pDispatch) = 0;
};
```

Creating the COM Object

The Automation API is responsible for creating the appropriate COM object for a given database-resident object. AutoCAD implements a set of interfaces for all database resident objects with corresponding Automation components. Many of these interfaces will be implemented automatically for your `AcDbObject`-derived or `AcDbEntity`-derived class when you use the ATL-based templates provided.

When creating extensions to the Automation API, you may need to create a COM object for a given `AcDbObjectId` or `AcDbObject` pointer. This can be done using `CoCreateInstance` followed by the use of `AcAxOleLinkManager` and `IAcadBaseObject` to set up the appropriate links. The following functions are exported for this purpose:

```
// Get the IUnknown of the existing COM object (or newly created COM
// object if one does not exist) that represents the AcDbObject
// passed in.
//
HRESULT
AcAxGetIUnknownOfObject(LPUNKNOWN* ppUnk, AcDbObjectId& objId,
    LPDISPATCH pApp);
HRESULT
AcAxGetIUnknownOfObject(LPUNKNOWN* ppUnk, AcDbObject* pObj,
    LPDISPATCH pApp);
```

COM objects are created via `CoCreateInstance()` using a `CLSID`, which identifies the object type. To retrieve the corresponding `CLSID` for a given `AcDbObject`-derived object, use its `getClassID()` function. This function is defined at the `AcDbObject` level and overridden at every other level in the class hierarchy that has a different COM object type to represent it.

```
// Get corresponding COM wrapper class ID.
//
virtual Acad::ErrorStatus getClassID(CLSID* pClsid) const;
```

For instance, if you create a custom entity (in other words, an `AcDbEntity`-derived class) and do not override `getClassID()`, then the `CLSID` returned is the one for `AcadEntity`. This means your custom entities will at least have base-level functionality even if you do not provide COM support for your entity.

There is an additional requirement for using the following APIs to create COM objects for your `AcDbObject`-derived class:

```
IAcadBlock::AddCustomObject(BSTR ClassName, LPDISPATCH* pObject)
IAcadModelSpace::AddCustomObject(BSTR ClassName,
    LPDISPATCH* pObject)
IAcadPaperSpace::AddCustomObject(BSTR ClassName,
    LPDISPATCH* pObject)
CAcadDictionary::AddObject(BSTR Keyword, BSTR ObjectName,
    IAcadObject** pObject)
```

These functions take the actual `AcDbObject`-derived class name (for example, `AcDbMyObject`) and create the COM object for you. After the COM object is created, `IAcadBaseObjectId::SetObjectId()` will be called on it to allow the `AcDbObject`-derived class to be instantiated and added to the database.

To obtain a `CLSID` for a given `AcDbObject`-derived class name, the system registry must contain an entry with the name of your `AcDbObject` and its corresponding `CLSID` value.

The registry layout looks like this:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\
    ObjectDBX\
        ActiveXCLSID\
            AcRxClassName\CLSID:REG_SZ:
                {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}
```

In the example above, replace `AcRxClassName` with the name of your `AcDbObject`-derived class (in other words, `AcDbMyObject`).

Implementation of Automation Objects

The following AutoCAD-specific interfaces are supported by the COM objects in the Automation API that represent an object in the database:

IAcadBaseObject

Maintains the link to an `AcDbObject` via an `AcDbObjectId`.

IAcadObjectEvents

Source interface that notifies COM clients when the `AcDbObject` has been modified.

IRetrieveApplication

Used to tell the COM object what to return for the Application property.

IAcadObject

Exposes all common properties and methods that apply to every object in the database.

IAcadEntity

Exposes all common properties and methods that apply to every entity in the database. (Only applicable for `AcDbEntity`-derived classes.)

The following interfaces are not AutoCAD specific, but are required for proper behavior:

IDispatch

Allows late binding. Browsers such as OPM require this interface.

IConnectionPointContainer

Used to keep a list of connection points.

IConnectionPoint

Used to allow COM clients to ask for notification.

ISupportErrorInfo

Informs COM clients that the object supports error info.

If you are creating a COM class to represent an `AcDbObject`-derived class, you will need to implement all of these interfaces.

ATL Templates

If you use ATL along with ATL-based templates from AutoCAD to create your Automation objects, all of the interfaces listed above will be implemented automatically. You can concentrate on the specific properties and methods for your `AcDbObject`-derived class; everything else is implemented by either Autodesk or Microsoft.

Autodesk provides the following ATL-based templates:

ATL-based templates (declared in *axtempl.h*)

Template	Implements
CProxy_AcadObjectEvents	IAcadObjectEvents, IConnectionPoint
IAcadBaseObjectImpl	IAcadBaseObject, IConnectionPointContainer
IRetrieveApplicationImpl	IRetrieveApplication
IAcadObjectDispatchImpl	IAcadObject, IDispatch
IAcadEntityDispatchImpl	IAcadEntity

By changing the derivation from the ATL `IDispatchImpl` template to `IAcadEntityDispatchImpl` or `IAcadObjectDispatchImpl`, you will have automatic implementation for all the required interfaces. The steps required to implement automation are covered in detail in "Interacting with AutoCAD."

Interacting with AutoCAD

User interaction (such as `acedGetPoint`) from an Automation call must be wrapped around a series of ObjectARX API calls. This allows you to save the AutoCAD "state" before the interaction and then restore it afterwards. It also ensures that any other out-of-process Automation requests are rejected for the duration of your interaction. This prevents another Automation client from changing the command line or database while you are waiting for user input.

ObjectARX APIs to use when interacting with the user include the following functions:

```
Adesk::Boolean acedSetOLELock(int handle, int flags=0);
Adesk::Boolean acedClearOLELock(int handle);
void acedPostCommandPrompt();
```

For example:

```
// Get a point in AutoCAD, even though the point is not used.
//
STDMETHODIMP CMyApp::GetPoint()
{
    // Establishing a lock informs AutoCAD to reject any other
    // out-of-process Automation requests. If this call is
    // made from an unknown context (e.g., not a normal AutoCAD
    // registered command or lisp), then it also saves the
    // current AutoCAD state.
    //
    if (acedSetOLELock(5) != Adesk::kTrue)
    {
        return E_FAIL;
    }

    // Do the input acquisition (interaction).
    //
    ads_point result;
    if(ads_getpoint(NULL, "Pick a point: ", result) != RTNORM)
    {
        return E_FAIL;
    }

    // Clear the lock to allow out-of-process Automation
    // requests to be accepted again. If the AutoCAD state was saved
    // during the call to acedSetOLELock(), then the saved state is
    // restored.
    //
    acedClearOLELock(5);

    // Forces AutoCAD to redisplay the command prompt.
    //
    acedPostCommandPrompt();
    return S_OK;
}
```

Document Locking

Automation requests can be processed in all of the possible AutoCAD contexts. This means you are responsible for locking a document before modifying it. There will also be times when you will want to make a document "current" temporarily. For example, when adding an entity to *MODELSPACE or *PAPERSPACE you need to lock and make the document current. Failure to lock the document in certain contexts will cause a "lock" violation during the modification of the database. Failure to make the document current will cause your entity to be "invisible" in the graphics display (even after a regen).

The ObjectARX API includes functions in a document manager class to do this. Since this is a common task, we have encapsulated the functionality into an exported class `AcAxDocLock`.

For example:

```
STDMETHODIMP CMyEntity::Modify()
{
    AcAxDocLock docLock(m_objId, AcAxDocLock::kNormal);
    if(docLock.lockStatus() != Acad::eOk)
    {
        return E_FAIL;
    }

    // It is now safe to modify the database
    //
    return S_OK;
}
```

Creating a Registry File

For your COM server to be fully functional, all components and their respective interfaces must be registered with the system. In addition, the type library must also be registered so that it can be used to implement `IDispatch` for your components.

The registry entries are typically created during installation of your software. Below you will find information to help create a registry file (*.reg*), which is highly useful and identifies the minimum amount of information required for your COM server.

General format (using *compoly.reg* as an example):

```
REGEDIT

; type library entries
HKEY_CLASSES_ROOT\TypeLib\{uuid of type library}
HKEY_CLASSES_ROOT\TypeLib\{uuid of type library}\1.0 =
    compoly 1.0 Type Library HKEY_CLASSES_ROOT\TypeLib\
    {uuid of type library}\1.0\HELPDIR = x:\some\path\to
HKEY_CLASSES_ROOT\TypeLib\{uuid of type library}\1.0\0\win32 =
    x:\some\path\to\compoly.tlb
HKEY_CLASSES_ROOT\TypeLib\{uuid of type library}\1.0\9\win32 =
    x:\some\path\to\compoly.tlb

; coclass entries
HKEY_CLASSES_ROOT\CLSID\{uuid of coclass} = ComPolygon Class
HKEY_CLASSES_ROOT\CLSID\{uuid of coclass}\InProcServer32 =
    x:\some\path\to\compoly.dll
```

```
; interface entries
HKEY_CLASSES_ROOT\Interface\{uuid of interface} =
    IComPolygon Interface
HKEY_CLASSES_ROOT\Interface\{uuid of interface}\TypeLib =
    {uuid of type library}
HKEY_CLASSES_ROOT\Interface\{uuid of interface}\ProxyStubClsid32 =
    {00020424-0000-0000-C000-000000000046}
```

The last two sections will repeat for every coclass and interface in your type library. The IDL file used to build the type library will contain all the uuids you need to fill in the blanks above. Below are commented excerpts from *compoly.idl* that identify each uuid.

```
[
    // uuid of type lib.
    //
    uuid(45C7F028-CD9A-11D1-A2BD-080009DC639A),
    version(1.0),
    helpstring("compoly 1.0 Type Library")
]
library COMPOLYLib
{
    // ... Code cut out for brevity.
    // IComPolygon interface
    [
        object,
        // uuid of interface
        //
        uuid(45C7F035-CD9A-11D1-A2BD-080009DC639A),
        dual,
        helpstring("IComPolygon Interface"),
        pointer_default(unique)
    ]
    interface IComPolygon : IAcadEntity
    {
        // ... Code cut out for brevity.
    };

    // ... Code cut out for brevity.
    // ComPolygon coclass
    [
        // uuid of coclass
        //
        uuid(45C7F036-CD9A-11D1-A2BD-080009DC639A),
        helpstring("ComPolygon Class"),
        noncreatable
    ]
    coclass ComPolygon
    {
        [default] interface IComPolygon;
        [source] interface IAcadObjectEvents;
    };
};
```

Exposing Automation Functionality

You can make functions, objects, or entities coded in ObjectARX available to developers who are accessing an ActiveX object model through VBA or some other programming environment.

Setting Up an ATL Project File

COM wrappers can be created as separate DLLs, or combined with your ObjectARX application. Both methods use the Microsoft ATL AppWizard to simplify the process. A separate DLL allows the system to manage resources more efficiently; it can release COM objects when they are not needed even if the ObjectARX application cannot be unloaded.

To set up a project for a COM wrapper in a separate DLL

1. Make sure *axauto15.dll*, which should be in the same directory as *acad.exe*, is in your search path.
2. From the Microsoft Visual C++ File menu, choose New.
3. Select ATL COM AppWizard on the Projects tab and enter a project name.
4. Choose the DLL server type. Additional project settings are optional.
5. Choose Finish and OK.
6. From the Insert menu or the Class view shortcut menu, choose New ATL Object.
7. Select a Simple Object in the Objects category and choose Next.
8. Enter a C++ Short Name on the Names tab; the Wizard will supply defaults for the remaining names.
9. On the Attributes tab, select Support IErrorInfo.
10. Choose OK.
11. From the Project menu, choose Settings.
12. On the C/C++ tab, choose C++ Language from the Category drop-down list and select Enable exception handling.
13. On the Link tab, add *axauto15.lib*, *oleaprot.lib*, and any other referenced ObjectARX libraries.
14. Choose OK.

To set up a project that combines a COM wrapper with an existing ObjectARX application

1. Make sure *axauto15.dll*, which should be in the same directory as *acad.exe*, is in your search path.
2. From the Microsoft Visual C++ File menu, choose New.
3. Select ATL COM AppWizard on the Projects tab and enter a project name.
4. Choose the DLL server type. Additional project settings are optional.
5. Choose Finish and OK.
6. Add all the CPP and H files from your ObjectARX application.
7. Update your include and library paths and DLL entry point as appropriate for an ObjectARX application.
8. Update DEF file by adding entry points. Change the DLL name to have an ARX extension.
9. At this point you should be able to compile to make sure that the ObjectARX application builds successfully.
10. From the Insert menu or the Class view shortcut menu, choose New ATL Object.
11. Select a Simple Object in the Objects category and choose Next.
12. Enter a C++ Short Name on the Names tab; the Wizard will supply defaults for the remaining names.
13. On the Attributes tab, select Support IErrorInfo.
14. Choose OK.
15. From the Project menu, choose Settings.
16. On the C/C++ tab, choose C++ Language from the Category drop-down list and select Enable exception handling.
17. On the Link tab, add *axauto15.lib*, *oleaprot.lib*, and any other referenced ObjectARX libraries.
18. Choose OK.

Writing a COM Wrapper

You can write a COM wrapper class to add select functionality to an ActiveX Automation model, or you can expose custom objects or entities.

Adding Functionality to an Object Model

In the simplest case, your COM wrapper class will expose one or more functions that you would like to make available to developers using programming environments such as VBA.

To create an Automation wrapper for an ObjectARX application

1. Set up your project according to the steps in "Setting Up an ATL Project File."
2. In the COM object header file, add `#include "axtempl.h"` (the main ActiveX Automation template header file).
3. If you want an application property, add the following entry to the COM_MAP:

   ```
   COM_INTERFACE_ENTRY(IRetrieveApplication)
   ```

4. In the IDL file, add `importlib ("c:\ACAD\acad.tlb");` after `importlib stdole32.tlb` and `importlib stdole2.tlb`. Make sure to use the correct path that matches your AutoCAD installation.
5. If the ObjectARX application and the COM wrapper are combined, add the following code to your main CPP file and call it `DllMain` in `AcRx::kInitAppMsg` and `AcRx::kUnloadAppMsg` with the appropriate parameters. This initializes the ATL object map, among other things.

   ```
   extern "C" HINSTANCE _hdllInstance;
   extern "C" BOOL WINAPI DllMain(HINSTANCE hInstance,
       DWORD dwReason,LPVOID /*lpReserved*/);
   ```

6. Build and register the application according to the steps in "Building and Registering a COM DLL."

Adding a Custom Object or Entity to an Object Model

If your COM wrapper class is encapsulating a custom object or entity, you will need to modify the ATL-generated code to support the appropriate interfaces. The directory *\docsamps\square* contains an example of a wrapper class for a custom entity written as a separate DLL.

To create an Automation wrapper for a custom object or entity

1. Set up your project according to the steps in "Setting Up an ATL Project File."
2. In the COM object header file, include *axtempl.h* (the main ActiveX Automation template header file) and the header file(s) for your custom objects or entities.
3. Change the derivation of the COM object or entity by removing the `IDispatchImpl` part of the derivation and replacing it with the following code:

   ```
   // For a custom object.
   //
   public IAcadObjectDispatchImpl<CWrapperClass,
       &CLSID_WrapperClass,IWrapperClass,
       &IID_IWrapperClass,&LIBID_LIBRARYLib>
   ```

```
// For a custom entity.
//
public IAcadEntityDispatchImpl<CWrapperClass,
    &CLSID_WrapperClass,IWrapperClass,
    &IID_IWrapperClass,&LIBID_LIBRARYLib>
```

4 Add the following entries to the COM_MAP:

```
COM_INTERFACE_ENTRY(IAcadBaseObject)
COM_INTERFACE_ENTRY(IAcadObject)
COM_INTERFACE_ENTRY(IAcadEntity) // For an entity only.
COM_INTERFACE_ENTRY(IRetrieveApplication)
COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer) // Only
                              // necessary to support events.
```

5 Add the following required override to the header file:

```
// IAcadBaseObjectImpl
//
virtual HRESULT
CreateNewObject(
    AcDbObjectId& objId,
    TCHAR* entryName,
    TCHAR* keyName);
```

This abstract function defined in the `IAcadBaseObjectImpl` template must be overridden to allow you to add default objects to the database.

6 Implement the `CreateNewObject()` function and any other object- or entity-specific functions.

The following example shows the implementation of `CreateNewObject()` from `AsdkSquareWrapper`:

```
HRESULT CAsdkSquareWrapper::CreateNewObject(
    AcDbObjectId& objId,
    AcDbObjectId& ownerId,
    TCHAR* keyName)
{
    try
    {
        //AXEntityDocLock(ownerId);
        Acad::ErrorStatus es;
        AcDbObjectPointer<AsdkSquare> pSq;
        if((es = pSq.create()) != Acad::eOk)
            throw es;
        AcDbDatabase* pDb = ownerId.database();
        pSq->setDatabaseDefaults(pDb);
        AcDbBlockTableRecordPointer
            pBlockTableRecord(ownerId, AcDb::kForWrite);
        if((es = pBlockTableRecord.openStatus()) != Acad::eOk)
            throw es;
        if((es = pBlockTableRecord->
            appendAcDbEntity(objId, pSq.object())) != Acad::eOk)
            throw es;
    }
```

```
    catch(const Acad::ErrorStatus)
    {
        //To become more sophisticated
        //
        return Error(L"Failed to create square",
            IID_IAsdkSquareWrapper, E_FAIL);
    }
    return S_OK;
}
```

7 In the IDL file, add `importlib("c:\ACAD\acad.tlb");` after `importlib stdole32.tlb` and `importlib stdole2.tlb`. Make sure to use the correct path that matches your AutoCAD installation.

8 Move the *acad.tlb* section to the top of the IDL file and move your custom object code so that it is within that section.

NOTE The IDL file modifications will cause the compiler to issue a warning stating that the interface does not conform. You can ignore this message.

---

9 Change the derivation in the IDL file from `IDispatch` to `IAcadObject` for a custom object or `IAcadEntity` for a custom entity.

10 In the section of the IDL file that corresponds to your wrapper coclass, add `[source] interface IAcadObjectEvents;` after the `[default]` line in order to support events. The IDL file will now appear similar to the following code:

```
import "oaidl.idl";
import "ocidl.idl";
[
    uuid(800F70A1-6DE9-11D2-A7A6-0060B0872457),
    version(1.0),
    helpstring("AsdkSquareLib 1.0 Type Library")
]
library ASDKSQUARELIBLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    importlib("v:\acad\acad2000\acad.tlb");
    [
        object,
        uuid(800F70AD-6DE9-11D2-A7A6-0060B0872457),
        dual,
        helpstring("IAsdkSquareWrapper Interface"),
        pointer_default(unique)
    ]
```

```
    interface IAsdkSquareWrapper : IAcadEntity
    {
        [propget, id(1), helpstring("property Number")]
            HRESULT Number([out, retval] short *pVal);
        [propput, id(1), helpstring("property Number")]
            HRESULT Number([in] short newVal);
    };
    [
        uuid(800F70AE-6DE9-11D2-A7A6-0060B0872457),
        helpstring("AsdkSquareWrapper Class")
    ]
    coclass AsdkSquareWrapper
    {
        [default] interface IAsdkSquareWrapper;
        [source] interface IAcadObjectEvents;
    };
};
```

11 After `#include <atlcom.h>` in *stdafx.h*, include *acad15.h* first, followed by any necessary ObjectARX header files.

12 At the end of *stdafx.cpp*, include *acad15_i.c*.

13 If the ARX application and the COM wrapper are combined, add the following code to your main CPP file and call it `DllMain` in `AcRx::kInitAppMsg` and `AcRx::kUnloadAppMsg` with the appropriate parameters. This initializes the ATL object map, among other things:

```
extern "C" HINSTANCE _hdllInstance;
extern "C" BOOL WINAPI DllMain(HINSTANCE hInstance,
    DWORD dwReason,LPVOID /*lpReserved*/);
```

14 Add the desired ActiveX methods and properties to your wrapper class by choosing Add Method or Add Property from the Class view Interface shortcut menu.

15 For any ObjectARX class being wrapped, override the `getClassId()` function for the custom object or entity with the following:

```
Acad::ErrorStatus
Class::getClassID(CLSID* pClsid) const
{
    *pClsid = CLSID_WrapperClass;
    return Acad::eOk;
}
```

16 In the file that contains the override for `getClassId()`, add:

```
#include <objbase.h>
#include <initguid.h>
#include "library_i.c" // File containing actual definitions of the
                       // IIDs and CLSIDs for the COM project.
```

17 Build and register the application according to the steps in "Building and Registering a COM DLL."

Building and Registering a COM DLL

A registry file that describes your application, type library, and Automation objects must be merged into the Windows system registry before the components in your DLL will be accessible. Some additional steps are required to link successfully a COM DLL and a separate ObjectARX application.

To prepare a COM DLL that is separate from the ObjectARX application

1. Build the COM DLL.
2. Add any decorated symbol names that appear in linker errors as unresolved external symbols to the Exports section of the ObjectARX DEF file.
3. Rebuild the ObjectARX application.
4. In the COM DLL, choose Settings from the Project menu.
5. On the Link tab, add the library for your ObjectARX application.
6. Choose OK.
7. Continue with the steps below to build and register your COM DLL.

To build and register your component server

1. Build the COM application.

   A message may appear indicating that REGSRVR32 failed to load AutoCAD. To eliminate this message in the future, remove all custom build steps for registering a COM server from your COM project settings.
2. Create a REG file for your application with GUIDs for the object and the type library that match those in the IDL file.

   The REG file must register the following information:

   - the object's `AcRxClass` name (so that `GetIUnknownOfObject` can locate the class)
   - the object's type library (so that when ATL calls `LoadRegTypeLib`, it will succeed)
   - the application name as the server for the COM object with that particular `CLSID` (so that `CoCreateInstance()` will work correctly)

The following example is the REG file for `AsdkSquareWrapper`:

```
REGEDIT
; This .REG file may be used by your SETUP program.
; If a SETUP program is not available, the entries below will be
; registered in your InitInstance automatically with a call to
; CWinApp::RegisterShellFileTypes and
; COleObjectFactory::UpdateRegistryAll.
HKEY_CLASSES_ROOT\TypeLib\{800F70A1-6DE9-11D2-A7A6-0060B0872457}
HKEY_CLASSES_ROOT\TypeLib\{800F70A1-6DE9-11D2-A7A6-0060B0872457}
    \1.1 = AsdkSquareLib 1.0 Type Library
HKEY_CLASSES_ROOT\TypeLib\{800F70A1-6DE9-11D2-A7A6-0060B0872457}
    \1.1\0\win32=
    E:\TEMP\square\AsdkSquareLib\Debug\AsdkSquareLib.dll
HKEY_CLASSES_ROOT\TypeLib\{800F70A1-6DE9-11D2-A7A6-0060B0872457}
    \1.1\9\win32 =
    E:\TEMP\square\AsdkSquareLib\Debug\AsdkSquareLib.dll
HKEY_CLASSES_ROOT\CLSID\{800F70AE-6DE9-11D2-A7A6-0060B0872457} =
    AsdkSquareWrapper Class
HKEY_CLASSES_ROOT\CLSID\{800F70AE-6DE9-11D2-A7A6-0060B0872457}
    \InProcServer32 =
    E:\TEMP\square\AsdkSquareLib\Debug\AsdkSquareLib.dll
HKEY_CLASSES_ROOT\Interface\{800F70AD-6DE9-11D2-A7A6-0060B0872457}
    = IAsdkSquareWrapper Interface
HKEY_CLASSES_ROOT\Interface\{800F70AD-6DE9-11D2-A7A6-0060B0872457}
    \TypeLib = {E3D2C633-69C9-11D2-A7A2-0060B0872457}
HKEY_CLASSES_ROOT\Interface\{800F70AD-6DE9-11D2-A7A6-0060B0872457}
    \ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}
```

3 Run the REG file from Explorer.

Object Property Manager API

The Object Property Manager (OPM) is a tool that allows users to view and modify properties of entities and objects easily. See the *AutoCAD User's Guide* for more information on the OPM features and user interface.

The OPM supports two kinds of properties. The first are properties that are defined for an object statically at compile time, called static properties. Dynamic properties, which are properties that can be added and configured at runtime, are also supported. For static properties, ObjectARX applications can provide the COM "wrapper" classes for their custom objects. The OPM will use these classes to determine which static properties are available for that object and how to "flavorize" those properties for display (such as, does the property require a custom combo box in the OPM, or need to bring up a dialog for property editing?). For dynamic properties, ObjectARX, VB, or VBA applications can create "property plug-ins" using the OPM Dynamic Properties API to configure and manage properties for any native or custom class at runtime.

AutoCAD COM Implementation

The OPM is essentially a control that parses type information from COM objects to determine their properties. When objects in the drawing are selected, the selection set is converted into an array of `IUnknown` pointers representing the COM objects that wrap all native entities in AutoCAD. These COM object wrappers are the fundamental support for the ActiveX Automation interface and are the underlying objects that the OPM communicates with.

These COM object wrappers implement `IDispatch` as well as other interfaces. `IDispatch` is the COM interface the OPM uses to get and set property data. It is also the native object representation in VB and VBA. To determine which properties are available for an object, the OPM calls `IDispatch::GetTypeInfo()`, which all AutoCAD COM wrappers implement. This function returns the type information for the object (an object that implements `ITypeInfo`). `ITypeInfo` is a standard Microsoft interface that wraps a data structure describing the methods and properties available on that object. Collections of type information used by VB and VBA to define the ActiveX object model are called type libraries.

The OPM takes property information, and based on the type of the property as it is defined in the IDL, constructs a property editor window appropriate for that type of property. For example, if the property type is numeric or textual, it constructs an edit box. If it is an enum, it creates a combo box with the enumerated value list. If it is a stock property such as Color, Layer, Linetype, Lineweight, or other built-in properties, it constructs the standard drop-downs for those that are the same as for the Object Property Toolbar (OPT).

The static type information for each COM object is not the only source of property information for the OPM. The OPM also queries the object for a few other interfaces to control things such as property categorization, property value names for drop-down lists, and instancing dialogs for per-property editing (such as the ellipsis button dialogs). These will be described in detail later in this section but will be referred to collectively as “flavoring” interfaces.

Static OPM COM Interfaces

If a custom object does not implement a COM object wrapper for itself, `GetIUnknownOfObject` will generate a default wrapper that implements the methods of `IAcadEntity` or `IAcadObject`, depending on if the underlying object can be cast to an `AcDbEntity`. The OPM then uses this object to display the Color, Layer, Linetype, and Lineweight properties, also known as the entity common properties. `ICategorizeProperties`, `IPerPropertyBrowsing`, and `IOPMPropertyExtension` are the flavoring interfaces.

This section describes the OPM flavoring interfaces in detail and explains how to use them to control the display of the static properties in the OPM. All the other static interfaces will be documented elsewhere as part of the Automation documentation.

ICategorizeProperties Interface

This interface is used by the OPM to categorize the properties shown in the control. It is optional but strongly recommended. If the object does not implement this interface, all properties are categorized under "General." The OPM does not support nesting of categories.

The OPM will use `QueryInterface` for this interface when it is collecting property information. Typically this will occur when the user selects objects, causing the pickfirst set to change. If the `QueryInterface` succeeds, it calls `MapPropertyToCategory` for each property defined by the type information for the object. If the category (`PROPCAT`) returned is not one of the predefined values, it calls `GetCategoryName` to determine which category to place the property in. If you are only interested in categorizing using the predefined values, you can return `E_NOTIMPL` from `GetCategoryName`. This requires that you know the `DISPID` for each of your properties. The Active Template Library (ATL) automatically assigns `DISPID` values to properties in the IDL files that define your interface. These are the numbers next to the "id" keyword in the property attribute list.

IPerPropertyBrowsing Interface

`IPerPropertyBrowsing` is a standard Microsoft interface. Please see the Microsoft documentation for a detailed explanation. It is typically used by property inspectors (such as the OPM) to display property pages for objects that have them. It has two basic functions. The first is to associate a property page or other dialog with a particular property via an ellipsis button on the

OPM dialog. The second purpose of `IPerPropertyBrowsing` is to support custom property drop-down lists in the OPM control.

IOPMPropertyExtension Interface

`IOPMPropertyExtension` is a collection of other flavoring functionality. `GetDisplayName` is used to override the name of a property from that in the type information. `Editable` is used to make properties that can be set in the type information to read-only in the OPM. `ShowProperty` is used to temporarily remove a property from being displayed in the OPM.

IOPMPropertyExpander Interface

The main purpose of this class is to allow one property to be broken out into several properties in the OPM. For example, Automation has a property called StartPoint for AcadLine. This property gets or sets a `VARIANT` that contains an array of doubles (technically the `VARIANT` contains a pointer to a `SAFEARRAY` of doubles) representing the start point of the line. This is somewhat more efficient and cleaner from an API point of view than having Automation properties called StartX, StartY, StartZ on AcadLine. However, OPM needs to display the properties expanded out in this fashion. In addition to splitting one property into an array of properties, you can also group the elements in that array. For example, for polyline vertices, there is one Automation property, "Coordinates," which returns an array of doubles, each successive pair representing the *X,Y* vertices of the 2D polyline. By specifying a grouping, the OPM will automatically create a spinner control for the property, allowing the user to enumerate and change the values of the vertices. These methods are optional, since in most cases you can create separate properties in the IDL.

Implementing Static OPM Interfaces

To implement COM object wrappers defining static properties for custom objects the easiest method is to use the ATL. The ATL makes it very easy to create COM objects that support `IDispatch`. The most difficult part is integrating the ObjectARX custom object code with the ActiveX Server DLLs that ATL generates.

Once the base object is working, it is easy to add properties that will show up in the OPM. See the previous section for instructions to create the base object COM wrapper for your custom objects.

To add properties

1. Go to Class View in the Visual C++ IDE, right-click on the custom entity interface (such as `IAsdkSquareWrapper`), and choose AddProperty.
2. For Property Type, choose Double. For Property Name, choose a property (such as SquareSize). Leave the parameters blank.
3. In the stub that the Wizard created for you, add the following query code (such as the `get_SquareSize` function from the polygon sample):

```
AcDbObjectPointer<AsdkSquare> pSq(m_objId, AcDb::kForRead);
if (pSq.openStatus() != Acad::eOk)
    return E_ACCESSDENIED;
double size;
pSq->squareSideLength(size);
*pVal = size;
return S_OK;
```

4. In the stub that the Wizard created, add the following modification code (such as the `put_SquareSize` function from the polygon sample):

```
AcDbObjectPointer<AsdkSquare> pSq(m_objId, AcDb::kForWrite);
if (pSq.openStatus() != Acad::eOk)
    return E_ACCESSDENIED;
pSq->setSquareSideLength(newVal);
return S_OK;
```

5. In AutoCAD, load the application (such as *squareui.arx*) and execute the command to create the custom entity.
6. Make sure OPM is loaded. Select the object. You should see and be able to change the entity common properties and the side length. Notice that SquareSize property displays under the "General" category.

To categorize properties

You may not want all your properties to show up under the "General" category, so this next section will demonstrate how to use the built-in categories.

1. Go to Class View in the Visual C++ IDE, right-click on the custom entity interface (such as `IAsdkSquareWrapper`), and choose AddProperty. Add properties for the square center and ID number.
2. Next change the derivation of the COM object to include `IOPMPropertyExtensionImpl` and `IOPMPropertyExpander`:

```
public IOPMPropertyExtensionImpl<CAsdkSquareWrapper>,
public IOPMPropertyExpander
```

3. Add the interfaces to the COM interface map:

```
COM_INTERFACE_ENTRY(IOPMPropertyExtension)
COM_INTERFACE_ENTRY(ICategorizeProperties)
COM_INTERFACE_ENTRY(IPerPropertyBrowsing)
COM_INTERFACE_ENTRY(IOPMPropertyExpander)
```

4 Add the declaration for the `IOPMPropertyExtension` interface:

```
// IOPMPropertyExtension
//
BEGIN_OPMPROP_MAP()
    OPMPROP_ENTRY(0, 0x00000001, PROPCAT_Data, \
        0, 0, 0, "", 0, 1, IID_NULL, IID_NULL, "")
    OPMPROP_ENTRY(0, 0x00000003, PROPCAT_Geometry, \
        0, 0, 0, "", 0, 1, IID_NULL, IID_NULL, "")
END_OPMPROP_MAP()
```

5 Add the following two inline functions to the class:

```
STDMETHOD(GetCategoryName)(
    THIS_
    /* [in]  */ PROPCAT propcat,
    /* [in]  */ LCID lcid,
    /* [out] */ BSTR* pbstrName)
    {return S_FALSE;}

virtual HINSTANCE GetResourceInstance()
{
    return _Module.GetResourceInstance();
}
```

6 Add the declarations for the following functions:

```
STDMETHOD(GetElementValue)(
    /* [in] */ DISPID dispID,
    /* [in] */ DWORD dwCookie,
    /* [out] */ VARIANT * pVarOut) ;
// Used for property expansion (currently variant types)
//
STDMETHOD(SetElementValue)(
    /* [in] */ DISPID dispID,
    /* [in] */ DWORD dwCookie,
    /* [in] */ VARIANT VarIn) ;
// Used for property expansion (currently variant types)
//
STDMETHOD(GetElementStrings)(
    /* [in] */ DISPID dispID,
    /* [out] */ OPMLPOLESTR __RPC_FAR *pCaStringsOut,
    /* [out] */ OPMDWORD __RPC_FAR *pCaCookiesOut) ;
//Used for property expansion (currently variant types)
//
STDMETHOD(GetElementGrouping)(
    /* [in] */ DISPID dispID,
    /* [out] */ short *groupingNumber) ;
// Used for property expansion (currently variant types)
//
STDMETHOD(GetGroupCount)(
    /* [in] */ DISPID dispID,
    /* [out] */ long *nGroupCnt) ;
```

```
STDMETHOD(GetPredefinedStrings)(
    /* [in] */ DISPID dispID,
    /* [out] */ CALPOLESTR *pCaStringsOut,
    /* [out] */ CADWORD *pCaCookiesOut);

STDMETHOD(GetPredefinedValue)(
    /* [in] */ DISPID dispID,
    /* [out] */ DWORD dwCookie,
    /* [out] */ VARIANT *pVarOut);
```

7 Add the implementation for the function in the CPP source file. These examples are for the `AsdkSquare` object:

```
STDMETHODIMP CAsdkSquareWrapper::GetElementValue(
    /* [in] */ DISPID dispID,
    /* [in] */ DWORD dwCookie,
    /* [out] */ VARIANT * pVarOut)
{
    if (pVarOut == NULL)
        return E_POINTER;
    AcDbObjectPointer<AsdkSquare> pSq(m_objId, AcDb::kForRead);
    if (pSq.openStatus() != Acad::eOk)
        return E_ACCESSDENIED;
    if (dispID == 0x03) {
        AcGePoint3d acgePt;
        pSq->squareCenter(acgePt);
        AcAxPoint3d acaxPt(acgePt);
        ::VariantCopy(pVarOut,&CComVariant(acaxPt[dwCookie]));
    }
    return S_OK;
}

STDMETHODIMP CAsdkSquareWrapper::SetElementValue(
    /* [in] */ DISPID dispID,
    /* [in] */ DWORD dwCookie,
    /* [in] */ VARIANT VarIn)
{
    AcDbObjectPointer<AsdkSquare> pSq(m_objId, AcDb::kForRead);
    if (pSq.openStatus() != Acad::eOk)
        return E_ACCESSDENIED;
    if (dispID == 0x03) {
        AcGePoint3d acgePt;
        pSq->squareCenter(acgePt);
        AcAxPoint3d acaxPt(acgePt);

        acaxPt[dwCookie] = V_R8(&VarIn);
        pSq->upgradeOpen();
        pSq->setSquareCenter(acaxPt);
    }
    return S_OK;
}

STDMETHODIMP CAsdkSquareWrapper::GetElementStrings(
    /* [in] */ DISPID dispID,
    /* [out] */ OPMLPOLESTR __RPC_FAR *pCaStringsOut,
    /* [out] */ OPMDWORD __RPC_FAR *pCaCookiesOut)
```

```
{
    if (dispID == 0x03)
    {
        long size;
        size = 3;
        pCaStringsOut->pElems =
            (LPOLESTR *)::CoTaskMemAlloc(sizeof(LPOLESTR) * size);
        pCaCookiesOut->pElems =
            (DWORD *)::CoTaskMemAlloc(sizeof(DWORD) * size);
        for (long i=0;i<size;i++)
            pCaCookiesOut->pElems[i] = i;

        pCaStringsOut->cElems = size;
        pCaCookiesOut->cElems = size;
        pCaStringsOut->pElems[0] = ::SysAllocString(L"Center X");
        pCaStringsOut->pElems[1] = ::SysAllocString(L"Center Y");
        pCaStringsOut->pElems[2] = ::SysAllocString(L"Center Z");
    }
    return S_OK;
}

STDMETHODIMP CAsdkSquareWrapper::GetElementGrouping(
    /* [in] */ DISPID dispID,
    /* [out] */ short *groupingNumber)
{
    return E_NOTIMPL;
}

STDMETHODIMP CAsdkSquareWrapper::GetGroupCount(
    /* [in] */ DISPID dispID,
    /* [out] */ long *nGroupCnt)
{
    return E_NOTIMPL;
}

STDMETHODIMP CAsdkSquareWrapper::GetPredefinedStrings(
    DISPID dispID, CALPOLESTR *pCaStringsOut,
    CADWORD *pCaCookiesOut)
{
    return E_NOTIMPL;
}
STDMETHODIMP CAsdkSquareWrapper::GetPredefinedValue(
    DISPID dispID, DWORD dwCookie, VARIANT *pVarOut)
{
    return E_NOTIMPL;
}
```

Dynamic Properties and OPM

The problem with type information is that it is static. It is defined at compile time in the *.idl* files and cannot be modified easily at runtime. Microsoft provides interfaces, most notably `ITypeInfo2` and `ICreateTypeInfo`, for so-called dynamic creation of type information. However, these interfaces only allow you to add type information from an existing `ITypeInfo` structure that refers to an existing dispatch interface. There is no runtime method for interrogating an object as to its property information. To fill this gap and allow any DLL to add properties to the OPM, the `IDynamicProperty` interface was defined. This allows you to implement an `IDynamicProperty` derived class for each property you wish to add to the OPM. The OPM can then call the methods of this class to get all the information it needs to display any type of property.

`IPropertyManager` controls how the OPM can get pointers to these property interfaces at runtime. For each `AcRxClass` object in AutoCAD, the client can get a pointer to an object that implements `IPropertyManager`. This is handled internally via protocol extensions. Once you have the property manager for the `AcRxClass` you are interested in, you can add your property classes to it via `IPropertyManager::AddProperty()`. When the user picks an object of that class, OPM will get the property manager for that class, enumerate all the property classes, and interrogate those classes for their property information, which it will then display along with that object's static properties. Note that the `IDynamicProperty` class makes no assumptions about where the property data is stored. It simply requires the `IDynamicProperty` implementer to provide it when `GetCurrentValueData()` is called. Similarly, when the user changes a dynamic property, OPM will call `SetCurrentValueData()` with the new value, leaving it up to the implementer to decide how to set that value. This leaves it up to you to decide how to make dynamic property data persistent.

The OPM uses `IPropertyManager` and `IDynamicProperty` not only for properties of objects, but also for displaying properties of the current space when no object is selected. For example, when no object is selected in the drawing, OPM needs to display properties relating to the UCS. Also, certain commands require the OPM to display property information (such as the ORBIT commands). These situations require defining special property managers for these specific "modes." Getting the property managers for the modes requires a slightly different mechanism than the procedure for getting the property managers for selectable objects. As mentioned earlier for properties of objects, there is a protocol extension for each class of object. This protocol extension object can be used by the developer to get the property

manager and add its property classes. For modal situations, there will be a set of predefined protocol extensions on the database that the developer can use to retrieve the property manager for that modal situation.

IDynamicProperty

As mentioned earlier, you should implement an instance of this class for each property that you wish to add to entities of a particular class.

AutoCAD DesignCenter COM API

In This Chapter

24

- AutoCAD DesignCenter API
- Registry Requirements for an AutoCAD DesignCenter Component
- Implementing the Interfaces for AutoCAD DesignCenter
- Customizing AutoCAD DesignCenter

AutoCAD has features that use the COM mechanism to query and modify objects. The AutoCAD DesignCenter (ADC) uses the COM mechanism to provide easily accessible drawing content. This chapter describes the COM interfaces that must be implemented by your application in order for it to participate and extend the AutoCAD DesignCenter.

AutoCAD DesignCenter API

AutoCAD DesignCenter provides an API that can be used to provide information about the content that it exposes. This API consists of four Component Object Model (COM) interfaces for management of contents. Two of these interfaces (`IAcDcContentBrowser` and `IAcDcContentView`) are designed to enable the component provider to display their content in AutoCAD DesignCenter, and the remaining two interfaces (`IAcDcContentFinder` and `IAcDcContentFinderSite`) are designed to enable the component provider to participate in the Finder mechanism of AutoCAD DesignCenter. The interfaces are described in the following sections. There is also another interface (`IAcPostDrop`) that component providers can implement to custom handle the right-click drag of items from AutoCAD DesignCenter.

IAcDcContentBrowser Interface

This interface is implemented in the AutoCAD DesignCenter framework and is used by the components to communicate get and set information. A pointer to this interface will be given to the components when their initialization method is called and the components are expected to cache this pointer to talk back to the framework.

This interface is similar to the `IShellBrowser` interface of the Windows namespace extension.

IAcDcContentView Interface

This interface is implemented by the components and is used by the AutoCAD DesignCenter framework to obtain content information from the component. A component that has registered itself as a content provider to AutoCAD DesignCenter would be queried for this interface at the appropriate time and will be asked to initialize itself. Once initialized, functions in this interface will be called at various times to get or set information in AutoCAD DesignCenter.

This interface is similar to the `IShellView` interface of the Windows namespace extension.

IAcDcContentFinderSite Interface

This interface is implemented in the AutoCAD DesignCenter framework and is used by the components to provide search results of a content type.

IAcDcContentFinder Interface

This interface is implemented by the components and is used by the AutoCAD DesignCenter framework to obtain search information from the components. A component that has registered itself as a content provider to AutoCAD DesignCenter would be queried for this interface at the appropriate time and will be asked to initialize itself. Once initialized, functions in this interface will be called at various times to get information appropriate for Finder dialog in the AutoCAD DesignCenter.

IAcPostDrop Interface

This interface is implemented by components and is used at the time of a right-click drag and drop of content entities.

Registry Requirements for an AutoCAD DesignCenter Component

In order for your component to be seen by the AutoCAD DesignCenter, it must be properly registered. This means it has to be included in AutoCAD DesignCenter's registry entries. To obtain the correct registry path, first get the initial AutoCAD path. This can be accomplished by using the `acrxProductKey()` function. Append *\AutodeskApps\AcadDC* to the initial path to complete the AutoCAD DesignCenter path. Under the AutoCAD DesignCenter path are keys for content provider applications. The following paragraphs describe the main keys and their specific subkeys.

Applications Key

This key is for content providers who want to register themselves and participate in AutoCAD DesignCenter custom mode.

Application Name

All information AutoCAD DesignCenter needs from a content provider participating in its custom mode is stored under this key. The name of the key would be the name of the application, as it appears in the custom mode. Under the Application Name key can be the following subkeys:

- Extensions

  This contains a list of extensions that a content provider supports (such as *.dwg*). Each key under extensions represents one extension. The name of the key is the name of the extension.

- Finder

  This key is optional. If an application wants to participate in the find functionality, then it is required to populate this key.

The following registry values demonstrate an applications registry branch:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\ACAD-
x:xxx\AutodeskApps\AcadDC\Applications\Autodesk\Finder\Drawings]
"LocalName"="Drawings"
"File Based Search"=dword:00000001
"Advanced Search"=dword:00000001
"Date Search"=dword:00000001
"Date Available"=dword:00000001
"Size Available"=dword:00000001
"Extension"=".dwg"
"Auto Append"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\ACAD-
x:xxx\AutodeskApps\AcadDC\Applications\Autodesk\Finder\Drawings
\Advanced Properties]
"Advanced Property1"="Block name"
"Advanced Property2"="Block and drawing description"
"Advanced Property3"="Attribute tag"
"Advanced Property4"="Attribute value"

[HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\ACAD-
x:xxx\AutodeskApps\AcadDC\Applications\Autodesk\Finder\Drawings
\Properties]
"Property1"="File Name"
"Property2"="Title"
"Property3"="Subject"
"Property4"="Author"
"Property5"="Keywords"
```

Extensions Key

This key is for content providers who want to register themselves and participate in AutoCAD DesignCenter desktop mode. These content providers handle only particular types of extensions, and they are not interested in participating in the custom mode setting of AutoCAD DesignCenter.

Extension Name

All information AutoCAD DesignCenter needs from a content provider participating in its desktop mode is stored under this key. An extension key could have any number of subkeys. The name of the subkey represents the name of the file extensions that are shown in the AutoCAD DesignCenter. There are two types of extensions stored in the registry, content type and container type. Under the Extension Name key can be the following subkeys:

- Content type

  If an extension key does not have any subkeys, it is considered to be a content type extension unless a content provider specifically sets the Container key to a value of one.

- Container type

  If an extension key does have subkeys, it is considered to be a container type extension. Values stored at this level of the key are ignored. A container key could have any number of subkeys. Each subkey represents a type of content that the container can handle.

The following registry values demonstrate an extensions registry branch:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\ACAD-
x:xxx\AutodeskApps\AcadDC\Extensions]

[HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\ACAD-
x:xxx\AutodeskApps\AcadDC\Extensions\.dwg]
"Default_CLSID"="{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}"
"Default_IconIndex"=dword:00000002
"Container"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R15.0\ACAD-
x:xxx\AutodeskApps\AcadDC\Extensions\.dwg\Blocks]
"CLSID"="{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}"
"IconIndex"=dword:00000000
"Prefix"="AcDc"
"LocalName"="Blocks"
```

CLASSID Registration

Minimum registration required by the component under `HKEY_CLASSES_ROOT` is as follows:

```
[HKEY_CLASSES_ROOT\CLSID\{<App CLSID xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx>}\InProcServer32]
@="<full path to application>\mycomponent.arx"
```

Implementing the Interfaces for AutoCAD DesignCenter

To provide the content in the AutoCAD DesignCenter, it is necessary to implement at least the `IAcDcContentView` interface. If the application is intended to participate in the Finder mechanism, then the application must support the `IAcDcContentFinder` interface as well. Implementing the `IAcDcContentFinder` interface is optional.

Typically, an application component would do the following:

- During the installation, it adds the appropriate entries to the registry.
- Call functions in `IAcDcContentBrowser` interface that are implemented by the AutoCAD DesignCenter framework. These are described in the following table:

IAcDcContentBrowser interface functions

Function	Description
AddPaletteItem	Adds an item in the AutoCAD DesignCenter palette.
AddNavigatorNode	Necessary only in case of custom view. Adds a node in the AutoCAD DesignCenter navigator.
GetDCFrameWindow	This is useful in the case of components wanting to have a valid window handle to display their context menus.
SetItemDescription	Sets the description text in description pane of the AutoCAD DesignCenter.
SetPaletteImageList	Sets the small/large image list used by the palette in the AutoCAD DesignCenter.
SetNavigatorImageList	Sets the small image list used by the navigator in the AutoCAD DesignCenter.
SetPaletteMultiSelect	Enables the option to select multiple content entities in the AutoCAD DesignCenter palette.
InsertPaletteColumn	Inserts a column (a subitem) for a content entity showing up in the palette.

IAcDcContentBrowser interface functions (*continued*)	
Function	**Description**
DeleteAllPaletteItems	Deletes all the items in the AutoCAD DesignCenter palette.
GetSelectedNavNodeText	Gets the selected navigator node text.
GetCurrentViewMode	Gets the current view mode (desktop, open drawings, etc.).
SetPaletteSubItem	Sets a subitem for an item in the AutoCAD DesignCenter palette.
SortPaletteItems	Sorts the items in the AutoCAD DesignCenter palette.

- Implement functions in `IAcDcContentView` interface in the component. These are described in the following table:

IAcDcContentView interface functions	
Function	**Description**
Initialize	Caches the given IAcDcContentBrowser. Optionally creates the component window and hides it. Optionally creates the component window and hides it. This allows a component to popup any context menus for its content. Creates at least one small and large image list. It also performs whatever other initialization is required.
NavigatorNodeClick	Shows the contents of the given file in the palette by calling AddPaletteItem() from the cached IAcDcContentBrowser. Optionally, it can ask the browser to sort the items. There are functions in IAcDcContentBrowser to do the sorting.
NavigatorMouseUp	Displays the context menu meaningful for a given container.
PaletteMouseUp	Displays the context menu meaningful for a given content entity/entities.
PaletteItemClick	Caches the given selected item and optionally sets the description of the item via the browser. This method is optional.

IAcDcContentView interface functions (*continued*)	
Function	**Description**
RenderPreviewWindow	Renders the image of the selected item on the given preview window. This method is optional.
PaletteItemDblClick	Shows the contents of the container in case the clicked item is a file. Does whatever is appropriate in AutoCAD if the clicked item is the content that the component is displaying.
PaletteColumnClick	Asks the content browser to do the sorting with the component's sort data. This method is optional.
PaletteBeginDrag	Begins the dragging of a content/container entity. The component is assumed to take care of the drag and drop of file/content types that are meaningful to it.
ReleaseBrowser	Releases the cached browser.
GetLargeImage	Gets the preview icon (32x32) for a given container file to show up in large icon view in Design Context.
QueryContextMenu	Appends the context menu items for a container (file) in the palette to a given context menu.
InvokeCommand	Invokes the command at a given command ID on the context menu that the component provided.

Customizing AutoCAD DesignCenter

It is relatively easy to create an application to provide custom content for AutoCAD DesignCenter using the ATL AppWizard. This example is available in the ObjectARX SDK, but it can be created from scratch following the steps below. The sample adds text files (with *.txt* extensions) as content. A single-click on the text file will show the contents in the description window (if it is active). A double-click will take the content of the file and insert it as mtext into model space at the center of the current viewport using the current text style.

Create an ActiveX Template Library Project

1 Create a new project in Visual C++ using the ATL COM AppWizard. For this example, we will name the project `AsdkDesignCenterSamp`. Chose the DLL server type and Support MFC. It is not required to use MFC, but for this sample, we will use it to make things easier. Click Finish and then OK to create the project.

2 Next add the necessary changes to make the project ObjectARX compatible. This includes adding the following entry code to the end of the *AsdkDesignCenterSamp.cpp* file:

```
extern "C" AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId)
{
    switch(msg)
    {
    case AcRx::kInitAppMsg:
        acrxRegisterAppMDIAware(appId);
        break;
    case AcRx::kUnloadAppMsg:
        break;
    default:
        break;
    }
    return AcRx::kRetOK;
}
```

3 You will also need to add the appropriate settings to the project and export the following symbols in the definition file (*.def*):

```
acrxEntryPoint
_SetacrxPtp
acrxGetApiVersion
```

Add Registry Support and a New ATL COM Object

1 Add the following registry initialization function to *AsdkDesignCenterSamp.cpp*. This function will set up the registry based on registry resources that will be added in a later step.

```
void registerAppInfo(HINSTANCE hInstance)
{
    USES_CONVERSION;
    HRESULT hRes = S_OK;
    CComPtr<IRegistrar> p;
    hRes = CoCreateInstance(CLSID_Registrar, NULL,
    CLSCTX_INPROC_SERVER, IID_IRegistrar, (void**)&p);
    if(SUCCEEDED(hRes))
```

```
    {
        // Get the AutoCAD Product key from the
        // registry into a CString.
        //
        CString csProdKey = acrxProductKey();

        // Use CStrings to obtain the authorization
        // stamp from the registry.
        //
        CString csPath = "SOFTWARE\\Autodesk\\AutoCAD\\R15.0\\";
        CString csStamp = csProdKey.Right(csProdKey.GetLength()
            - csPath.GetLength());

        _TCHAR szRegKey[_MAX_PATH];
        _tcscpy(szRegKey, csStamp);
        LPOLESTR pszId = T2OLE("AUTH");

        // do a runtime swap of the registry key value.
        //
        p->AddReplacement(pszId, T2OLE(szRegKey));

        _TCHAR szModule[_MAX_PATH];
        GetModuleFileName(hInstance, szModule, _MAX_PATH);
        LPCOLESTR szType = OLESTR("REGISTRY");
        LPOLESTR pszModule = T2OLE(szModule);

        // Pull the registry entries from the resource ID.
        //
        hRes = p->ResourceRegister(pszModule, IDR_REGISTRY1,
             szType);
        if(FAILED(hRes))
          AfxMessageBox("Error registering the app info.");
    }
}
```

2. Now add a new ATL Object that will support the `IAcDcContentView` interface. In Visual C++, choose Insert, New ATL Object. In the dialog pick Objects and choose Simple Object. Click Next and enter the name for the ATL Object. For this example, call it `AsdkDcContent`. Now choose the Names tab and click support `ISupportErrorInfo`. Click OK to create the object.
3. Next we need to add some registry information to the resource section of the project. First create a new file called *AsdkDesignCenterSamp.rgs*. The following listing should be changed for your specific project, where the class ID (`CLSID`) should be copied from your IDL file. Use the `CLSID` that corresponds to the `IAsdkDcContent` interface. Since these are `GUID` values, they are different for each new project. Also for other projects, you will need to change the extensions sections and also add the name of your specific class. Again, this example uses `AsdkDcContent`.

```
HKLM
{
  NoRemove 'SOFTWARE'
  {
    NoRemove 'Autodesk'
    {
      NoRemove 'AutoCAD'
      {
        NoRemove 'R15.0'
        {
          NoRemove '%AUTH%'
          {
            NoRemove 'AutodeskApps'
            {
              NoRemove 'AcadDC'
              {
                NoRemove 'Extensions'
                {
                  ForceRemove '.txt'
                  {
                    val CLSID = s '{<Your CLSID>}'
                    IconIndex = d '0'
                  }
                }
                NoRemove 'Applications'
                {
                  ForceRemove 'AsdkDcContent'
                  {
                    'Extensions'
                    {
                      .txt
                      {
                        val CLSID = s '{<Your CLSID>}'
                        val IconIndex = d '0'
                      }
                    }
                    CustomView = s 'Yes'
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Save this file and in Visual C++ bring the ResourceView forward. Open the resources listing and expand the "REGISTRY" node. Right click and import the registry file. It should give the resource an ID of `IDR_REGISTRY1`, but if it does not, rename it so that it matches the call in `registerAppInfo`.

Add Code to Support the New ATL COM Object

1 Now we must add code to support the new interface. First add an include for the *dcapi.idl* file to the *AsdkDesignCenterSamp.idl*. This include should be made after the two default imports:

```
import "oaidl.idl";
import "ocidl.idl";
#include "dcapi.idl"
```

2 Open the *AsdkDcContent.h* header file and change the derivation for the new class to include `CWindowImplBase` and `IAcDcContentView` as follows:

```
class ATL_NO_VTABLE CAsdkDcContent :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAsdkDcContent, &CLSID_AsdkDcContent>,
    public ISupportErrorInfo,
    public IDispatchImpl<IAsdkDcContent, &IID_IAsdkDcContent,
        &LIBID_ASDKDESIGNCENTERSAMPLib>,
    public CWindowImplBase,
    public IAcDcContentView
{
```

3 Now enter the object's interfaces into the COM map using the `COM_INTERFACE_ENTRY` macro:

```
BEGIN_COM_MAP(CAsdkDcContent)
    COM_INTERFACE_ENTRY(IAsdkDcContent)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(ISupportErrorInfo)
    COM_INTERFACE_ENTRY(IAcDcContentView)
END_COM_MAP()
```

4 Add a message map to the class:

```
BEGIN_MSG_MAP(CAsdkDcContent)
END_MSG_MAP()
```

5 Add the following declarations for the `IAcDcContentView` interface and some utility methods:

```
public:
    void OpenAndDisplayTextFile();
    void OpenAndInsertTextFile();

    CString OpenAndReadTextFile(DWORD& length);

    STDMETHOD(Initialize)(/*[in]*/ VARIANT varBrowser);
    STDMETHOD(SetImageLists)();
    STDMETHOD(NavigatorNodeExpanding)(
          /*[in]*/ VARIANT varhNode
        , /*[in]*/ BSTR bstrFullPath);
    STDMETHOD(NavigatorNodeCollapsing)(
          /*[in]*/ VARIANT varhNode
        , /*[in]*/ BSTR bstrFullPath);
    STDMETHOD(NavigatorNodeClick)(
          /*[in]*/ VARIANT varhNode
```

```
        , /*[in, string]*/ BSTR bstrFullPath);
    STDMETHOD(NavigatorMouseUp)(
          /*[in]*/ VARIANT varhNode
        , /*[in, string]*/ BSTR bstrFullPath
        , /*[in]*/ VARIANT varX
        , /*[in]*/ VARIANT varY);
    STDMETHOD(PaletteItemClick)(/*[in]*/ BSTR bstrItemText);
    STDMETHOD(PaletteItemDblClick)(/*[in]*/ BSTR bstrItemText);
    STDMETHOD(PaletteColumnClick)(/*[in]*/ VARIANT varIndex);
    STDMETHOD(PaletteMouseUp)(
          /*[in]*/ VARIANT varButton
        , /*[in]*/ VARIANT varItemTexts
        , /*[in]*/ VARIANT varX
        , /*[in]*/ VARIANT varY);
    STDMETHOD(PaletteMouseDown)(
          /*[in]*/ VARIANT varButton
        , /*[in]*/ BSTR bstrFullText
        , /*[in]*/ VARIANT varX
        , /*[in]*/ VARIANT varY);
    STDMETHOD(RenderPreviewWindow)(
          /*[in]*/ BSTR bstrFullText
        , /*[in]*/ VARIANT varhPreviewWindow);
    STDMETHOD(PreviewMouseUp)(
          /*[in]*/ VARIANT varButton
        , /*[in]*/ VARIANT varX
        , /*[in]*/ VARIANT varY);
    STDMETHOD(Refresh)();
    STDMETHOD(PaletteBeginDrag)(
          /*[in]*/ VARIANT varItemTexts
        , /*[in]*/ VARIANT varX
        , /*[in]*/VARIANT varY);
    STDMETHOD(ReleaseBrowser)();
    STDMETHOD(QueryContextMenu)(
          /*[in]*/ VARIANT varhMenu
        , /*[in]*/ VARIANT varIndex
        , /*[in]*/ VARIANT varCmdFirst
        , /*[in]*/ VARIANT varCmdLast
        , /*[in]*/ VARIANT varItemTexts);
    STDMETHOD(InvokeCommand)(/*[in]*/ VARIANT varMenuID);
    STDMETHOD(IsExpandable)(
          /* [string][in] */ BSTR bstrItemText
        , /* [retval][out] */ VARIANT __RPC_FAR
        *pvarIsExpandable);
    STDMETHOD(GetLargeImage)(
          /* [in] */ BSTR bstrFileName
        , /* [out][in] */ VARIANT __RPC_FAR
        *pvarhLargeImage);
    STDMETHOD(GetSmallImageListForContent)(
          /*[in]*/ BSTR bstrFileName
        , /*[out]*/ VARIANT *pvarhImageList);
    STDMETHOD(GetLargeImageListForContent)(
          /*[in]*/ BSTR bstrFileName
        , /*[out]*/ VARIANT *pvarhImageList);
private:
    char * m_strSelectedItemText;
    IAcDcContentBrowser* m_pBrowser;
```

6 Add the code to implement the methods just added. Note that many of these methods do nothing but are necessary to complete the interface. The example makes use of the single-click (`PaletteItemClick`) and double-click (`PaletteItemDblClick`) events.

```
Adesk::Boolean append(AcDbEntity* pEntity)
{
    AcDbBlockTable *pBlockTable;
    AcApDocument* pDoc = acDocManager->curDocument();
    Acad::ErrorStatus es = acDocManager->lockDocument(pDoc);
    if (es != Acad::eOk) {
        acedAlert("Failed to lock the document!");
        return Adesk::kFalse;
    }

    AcDbDatabase* pDb = pDoc->database();

    es = pDb->getBlockTable(pBlockTable, AcDb::kForRead);
    if (es != Acad::eOk) {
        acedAlert("Failed to get block table!");
        return Adesk::kFalse;
    }
    AcDbBlockTableRecord *pBlockRec;
    es = pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockRec,
        AcDb::kForWrite);
    if (es != Acad::eOk) {
        acedAlert("Failed to get block table record!");
        pBlockTable->close();
        return Adesk::kFalse;
    }
    es = pBlockRec->appendAcDbEntity(pEntity);
    if (es != Acad::eOk) {
        acedAlert("Failed to append entity!");
        pBlockTable->close();
        pBlockRec->close();
        delete pEntity;
        return Adesk::kFalse;
    }
    pBlockRec->close();
    pBlockTable->close();

    acDocManager->unlockDocument(pDoc);
    return Adesk::kTrue;
}

STDMETHODIMP CAsdkDcContent::Initialize(VARIANT varBrowser)
{
    m_pBrowser = (IAcDcContentBrowser*)varBrowser.punkVal;
    m_pBrowser->AddRef();
    return S_OK;
}
```

```
STDMETHODIMP CAsdkDcContent::SetImageLists()
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::NavigatorNodeExpanding(
    VARIANT varhNode,
    BSTR bstrFullPath)
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::NavigatorNodeCollapsing(
    VARIANT varhNode,
    BSTR bstrFullPath)
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::NavigatorNodeClick(
    VARIANT varhNode,
    BSTR bstrFullPath)
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::NavigatorMouseUp(
    VARIANT varhNode,
    BSTR bstrFullPath,
    VARIANT varX,
    VARIANT varY)
{
    return S_OK;
}

CString CAsdkDcContent::OpenAndReadTextFile(DWORD& length)
{
    CFile fileText;
    fileText.Open(m_strSelectedItemText, CFile::modeRead);
    length = fileText.GetLength();
    char *strBuff = new char[length];
    fileText.Read(strBuff, length);
    fileText.Close();
    CString cstrBuff(strBuff);
    delete strBuff;
    cstrBuff.SetAt(length, '\0');
    cstrBuff.FreeExtra();
    return cstrBuff;
}

void CAsdkDcContent::OpenAndDisplayTextFile()
```

```
{
    DWORD length;
    CString cstrBuff = OpenAndReadTextFile(length);
    BSTR bstrBuf = cstrBuff.AllocSysString();
    m_pBrowser->SetItemDescription(bstrBuf);
    ::SysFreeString(bstrBuf);
}

STDMETHODIMP CAsdkDcContent::PaletteItemClick(BSTR bstrItemText)
{
    USES_CONVERSION;
    m_strSelectedItemText = OLE2T(bstrItemText);
    OpenAndDisplayTextFile();
    return S_OK;
}

void CAsdkDcContent::OpenAndInsertTextFile()
{
    DWORD length;
    CString cstrBuff = OpenAndReadTextFile(length);
    cstrBuff.Replace("\015\012", "\\P");

    struct resbuf resbufViewCtr;
    resbufViewCtr.restype = RT3DPOINT;
    acedGetVar("VIEWCTR", &resbufViewCtr);
    AcGePoint3d pt(resbufViewCtr.resval.rpoint[X],
        resbufViewCtr.resval.rpoint[Y],
        resbufViewCtr.resval.rpoint[Z]);

    AcDbMText *pMText = new AcDbMText();
    pMText->setLocation(pt);
    pMText->setContents(cstrBuff);
    append(pMText);
    pMText->downgradeOpen();
    pMText->draw();
    pMText->close();
}

STDMETHODIMP CAsdkDcContent::PaletteItemDblClick(
    BSTR bstrItemText)
{
    USES_CONVERSION;
    m_strSelectedItemText = OLE2T(bstrItemText);
    OpenAndInsertTextFile();
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::PaletteColumnClick(
    VARIANT varIndex)
{
    return S_OK;
}
```

```
STDMETHODIMP CAsdkDcContent::PaletteMouseUp(
    VARIANT varButton,
    VARIANT varItemTexts,
    VARIANT varX,
    VARIANT varY)
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::PaletteMouseDown(
    VARIANT varButton,
    BSTR bstrFullText,
    VARIANT varX, VARIANT varY)
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::RenderPreviewWindow(
    BSTR bstrFullText,
    VARIANT varhPreviewWindow)
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::PreviewMouseUp(
    VARIANT varButton,
    VARIANT varX,
    VARIANT varY)
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::Refresh()
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::PaletteBeginDrag(
    VARIANT varItemTexts,
    VARIANT varX,
    VARIANT varY)
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::ReleaseBrowser()
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::QueryContextMenu(
    VARIANT varhMenu,
    VARIANT varIndex,
    VARIANT varCmdFirst,
    VARIANT varCmdLast,
    VARIANT varItemTexts)
```

```
{
    return S_OK;
}

STDMETHODIMP CAsdkDcContent::InvokeCommand(VARIANT varMenuID)
{
    return S_OK;
}

STDMETHODIMP
CAsdkDcContent::IsExpandable(
    /* [string][in] */ BSTR bstrItemText,
    /* [retval][out] */ VARIANT __RPC_FAR *pvarIsExpandable)
{
    pvarIsExpandable->iVal = TRUE;
    return S_OK;
}

STDMETHODIMP
CAsdkDcContent::GetLargeImage(
    /* [in] */ BSTR bstrFileName,
    /* [out][in] */ VARIANT __RPC_FAR *pvarhLargeImage)
{
    return E_NOTIMPL;
}

STDMETHODIMP
CAsdkDcContent::GetSmallImageListForContent(
    BSTR bstrFileName,
    VARIANT *pvarhImageList)
{
    return E_NOTIMPL;
}

STDMETHODIMP
CAsdkDcContent::GetLargeImageListForContent(
    BSTR bstrFileName,
    VARIANT *pvarhImageList)
{
    return E_NOTIMPL;
}
```

7 Now include the appropriate header files in the *sdtafx.h* file. You will also need to add a definition to undefine _DEBUG, since the AutoCAD libraries are non-debug. Here is what the file should look like:

```
#if defined(_DEBUG) && !defined(ARX_DEBUG)
#undef _DEBUG
#define ARX_DEBUG
#endif
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#define STRICT
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0400
#endif
#define _ATL_APARTMENT_THREADED
#include <afxwin.h>
#include <afxdisp.h>
#include <atlbase.h>
// You may derive a class from CComModule and use
// it if you want to override something, but do not
// change the name of _Module.
//
extern CComModule _Module;
#include <atlcom.h>

#include <atlwin.h>
#include <adslib.h>
#include <dbmain.h>
#include <dbsymtb.h>
#include <dbmtext.h>
#include <acdocman.h>
#include <aced.h>
#include <rxregsvc.h>

#ifdef ARX_DEBUG
#undef ARX_DEBUG
#define _DEBUG
#endif
```

Part VI

ObjectARX Libraries

The ObjectDBX Libraries

25

In This Chapter

- Introduction
- Using ObjectDBX
- Differences between ObjectDBX and ObjectARX
- Localization and XMX Files
- Transaction Management
- Creating a Viewer
- Demand Loading
- Installing the ObjectDBX Libraries
- Tips and Techniques
- Known Limitations

ObjectDBX is the successor to DWG Unplugged, and this chapter describes the changes and enhancements that the ObjectDBX SDK provides, along with a description of how to implement applications using ObjectDBX.

Introduction

The ObjectDBX SDK is the interface among host applications, drawing (*.dwg*) files, custom application (*.arx*) files, and custom object (*.dbx*) files.

Overview

ObjectDBX comprises a set of DLLs that can be used to implement custom objects contained in an AutoCAD 2000 drawing file, and to implement applications that manipulate DWG files without the presence of AutoCAD. Part of this capability was formerly presented in the DWG Unplugged product, but the ObjectDBX SDK replaces and goes beyond the DWG Unplugged technology by providing the support necessary for intelligent object systems. The ObjectDBX SDK allows you to create non-AutoCAD host applications that can read and write DWG files.

Host Applications

A "host application" is one that contains a `main()`, `WinMain()`, or `dllMain()` function in its code, and provides the host services that an ObjectDBX or ObjectARX program needs.

There are two types of host applications that can take advantage of the interface ObjectDBX provides. One type of host application is AutoCAD 2000, with or without associated ObjectARX applications. The second type is a non-AutoCAD host application. A non-AutoCAD host application cannot load an ObjectARX application, and can only take advantage of the specific interfaces provided by the DLLs contained in ObjectDBX.

ObjectDBX Libraries

ObjectDBX libraries contain intelligence that enables custom objects (geometry, components, non-graphic objects, and so on) to operate as extensions, or custom objects, inside AutoCAD. The files that implement these objects are given the extension *.dbx*, which stands for DataBase eXtension. A DBX file is basically an ObjectARX application that has been written to work with the ObjectDBX libraries instead of with the ObjectARX (AutoCAD) libraries.

User Interface and Database Access

ObjectDBX allows you to write separate binaries for the user-interface (UI) and database (DB) portions of your application. The file containing the UI component would have the extension *.arx* and would contain the code that issues prompts, displays dialogs, modifies menus in AutoCAD, and so forth. The DB component file would have the extension *.dbx* and would contain the code that implements your custom objects by creating them, displaying them, transforming them, and so on. If your application is separated into user-interface and database portions, your custom objects will still be handled properly without the ObjectARX application that provided the user interface being present.

For example, suppose you implement a custom object called "Sink," and that the code to display and modify Sink is in *sink.dbx*, while the code to prompt the user for Sink creation values is in *sink.arx*. Your user can load *sink.arx* from AutoCAD and use it (with *sink.dbx*, which will be loaded automatically) to create a custom sink in a drawing. Later, that drawing can be loaded by any other host application (including AutoCAD), and if the user has a copy of the *sink.dbx* file available, the Sink objects will display properly, instead of as proxies.

Using ObjectDBX

Developing applications with ObjectDBX is very similar to developing applications with ObjectARX. The C++ API found in the class hierarchy of ObjectARX is largely the same in ObjectDBX. Your principal task as a developer is to understand exactly what subset of ObjectARX is at your disposal.

Getting Started with ObjectDBX

The following are some general items you should be aware of before using ObjectDBX.

C Runtime Libraries

The release DLLs for ObjectDBX are linked to the release versions of the Microsoft C Runtime library, *MSVCRT.dll*. Your application should also link with the release version and not the debug version. Mixing release and debug or static and DLL combinations of the C Runtime libraries may cause memory allocation or deallocation errors.

Multithreading

While the ObjectDBX DLLs support multiple processes on Windows NT and Windows 95, they have not been made thread safe.

AcDbDatabase

Always instantiate an `AcDbDatabase` before using any AcDb functions. Refer to "Always Instantiate an AcDbDatabase" on page 676.

ObjectDBX Library Changes

Many libraries have been renamed to include their version number. In addition, there are several new libraries. The following table cross-references the new and old library names.

ObjectDBX libraries

Release 14 Library Name	AutoCAD 2000 Library Name
acfirst.dll	ac1st15.dll
ism.lib	acISMobj15.lib
libacge.lib	acge15.lib
libacgex.lib	acgex15.lib
libacbr.lib	acbr15.lib
Not present	achapi15.lib
Not present	acdb15.lib
Not present	acrx15.lib
Not present	acutil15.lib

When linking host applications, be sure to link *acdb15.lib* first, *rxapi.lib* second, and any other libraries afterwards.

The Application Services Class

When ObjectDBX is used to create a host application, the code in the ObjectDBX library expects the host application to provide it with certain services; for example, a file find mechanism. When you write an ObjectDBX

host application, you are required to implement these services, which will be used by both ObjectDBX itself, and potentially by other DBX applications. These settings and services are made available by the application object class `AcDbHostApplicationServices`. Your host application must derive, create, and register an instance of this class with ObjectDBX, which in turn invokes member functions of the class instance as needed.

The header file for application services is *dbapserv.h*. The classes and methods in this header file fall into one of three categories:

- Those you *must* override, because no default implementation is provided as the method is assumed to be very application-specific. These are declared to be pure virtual.
- Those you *may* override, but that have a default implementation that will minimally satisfy the database code. These are declared virtual.
- Those you *may not* override, as they are expected to operate identically in all host applications. These are generally not declared virtual.

It is required that any ObjectDBX host application must provide a class derived from `AcDbHostApplicationServices`. This is different from the way DWG Unplugged worked, where a default service was provided. A detailed description of the class exists in the *ObjectARX Reference*, where each method is described with its default implementation (if it has one), what you must do to override the method successfully, and how to call the method. When your application is initializing, it should create an instance of your class derived from `AcDbHostApplicationServices`. Configure it as necessary and make the object available to the application by calling the global function `acdbSetHostApplicationServices()`.

Differences between ObjectDBX and ObjectARX

The most visible difference between ObjectDBX and ObjectARX applications is the presence of AutoCAD. ObjectDBX can be used to create DBX files that will run with any host application, but ObjectARX applications always require AutoCAD as their host application. ObjectDBX can also be used to create host applications, but ObjectARX cannot be used to create host applications.

The ObjectDBX and ObjectARX libraries share most of the same C++ code base used to develop AutoCAD. However, ObjectDBX does not use ObjectARX functions specific to an AutoCAD drawing session, and some of

those functions have replacements in ObjectDBX that make up for the runtime absence of AutoCAD.

The sample programs included with ObjectDBX illustrate various uses of the ObjectDBX libraries. The *creatent* sample is the simplest and best place to begin studying ObjectDBX application development.

The following sections provide detailed information on differences between the ObjectARX and ObjectDBX libraries.

AcEditorReactor Class

The following `AcEditorReactor` notifications are valid in ObjectDBX:

- `dwgFileOpened`
- `databaseToBeDestroyed`
- `saveComplete`
- `beginInsert`
- `otherInsert`
- `abortInsert`
- `endInsert`
- `wblockNotice`
- `beginWblock`
- `otherWblock`
- `endWblock`
- `beginDeepClone`
- `beginDeepCloneXlation`
- `abortDeepClone`
- `endDeepClone`
- `sysVarChanged`
- `sysVarWillChange`

AcGi API

The intended use of the AcGi layer is described in "AcGi" on page 664. The AutoCAD-specific implementation of its AcGi layer is not part of ObjectDBX. Instead, ObjectDBX provides its own graphics interface for displaying AutoCAD entities.

Localization and XMX Files

Since *acdb.xmx* is translated for all localized versions of AutoCAD, pretranslated XMX files are now shipped as part of the ObjectDBX SDK, so you can create a fully language-localized end product. This will allow you to create several language versions of your application, or your application can query the user with a choice of several languages.

Choosing the language will be the responsibility of your application. To that end, the loading of the XMX file is in the function `acdbValidateSetup()`. This function takes an LCID parameter to specify the application's choice of language. The function will attempt to load that XMX file first by using the `AcDbHostApplicationServices::findFile()` method and, if that fails, by looking in the directory that contains *AcDb15.dll*.

The signature for `acdbValidateSetup()` is

```
Acad::ErrorStatus
acdbValidateSetup(
    long lcid);
```

The *acdb.xmx* file is now named *acdbLLL.xmx*, where *LLL* is the three-letter language-localization abbreviation, which can be derived from the LCID.

Autodesk supports, and will eventually ship or otherwise provide, *acdbLLL.xmx* files in the following languages.

XMX file types

Language	Language Abbreviation	Language ID from LCID
English (USA)	ENU	0409
Chinese (Taiwan)	CHT	0404
Chinese (Simplified)	CHS	0804
Czech	CSY	0405
French (Default)	FRA	040c
German (Default)	DEU	0407
Greek	ELL	0408
Hungarian	HUN	040e
Italian	ITA	0410

XMX file types (*continued*)		
Language	**Language Abbreviation**	**Language ID from LCID**
Japanese	JPN	0411
Korean	KOR	0412
Polish	PLK	0415
Portuguese (Brazilian)	PTB	0416
Portuguese (Default)	PTG	0816
Russian (Default)	RUS	0419
Spanish (Default)	ESP	040a

As an ObjectDBX developer, you must do two things to create a language-localized end-product:

- You must ship the appropriate *acdbLLL.xmx* files along with your product.
- You must inform ObjectDBX which *acdbLLL.xmx* file to load, by passing the appropriate LCID to `acdbValidateSetup()`.

If the LCID does not correspond to one of the three-letter abbreviations above, or if the appropriate XMX file was not shipped, your ObjectDBX application will fail to load properly.

If it is unable to find the desired *acdb.xmx* file, `acdbValidateSetup()` will attempt to load English as a default. Again, it will first use `findFile()`, and next assume the same path as *AcDb15.dll*. If it finds English, but English was not the requested language, `Acad::eFileNotFound` is returned. If the function is unable to find any *acdb.xmx* file, it will halt with `fatalError()`, and your application will not load.

Transaction Management

Transaction handling is now part of ObjectDBX instead of AutoCAD, and the corresponding library is *acdb.dll* instead of the AutoCAD executable. There is one new class, `AcDbTransactionManager`, as part of this change.

AcTransaction and AcTransactionReactor Classes

These classes have been moved from AutoCAD to the ObjectDBX DLL. Their header file is now *dbtrans.h*, though you may continue to use the *actrans.h* header since *actrans.h* includes *dbtrans.h*. For information about the classes, please see the sections on `AcTransaction` and `AcTransactionReactor` in the *ObjectARX Reference*.

AcTransactionManager and AcDbTransactionManager Classes

The `AcDbTransactionManager` class is new in this release, and the existing `AcTransactionManager` class is now derived from `AcDbTransactionManager`. All the methods of `AcTransactionManager`, except `enableGraphicsFlush()` and `flushGraphics()`, now belong to `AcDbTransactionManager`. The `enableGraphicsFlush()` and `flushGraphics()` methods are still members of the `AcTransactionManager` class. For a description of the classes and methods, please see the *ObjectARX Reference*. The `AcTransactionManager` class is still part of the *acad.lib* library.

Creating a Viewer

ObjectDBX includes components that can be used to develop a viewer for geometric models stored in an AutoCAD database. These components work together to form a complete viewing library but may be used or replaced independently by developers. The components interact with AcDb models through the AcGi API, which is the same interface that the AutoCAD graphics system uses to interact with AcDb.

Viewer Components

ObjectDBX provides three distinct tools that work together to implement a viewer:

- The AcGix elaboration library
- The SimpleView sample vector taker using HDC
- The WhipView display list vector taker

The sample ObjectDBX application ViewAcDb demonstrates the use of these components. It is supplied in binary form and in full source form with a Microsoft Visual C++ project file.

Although it is not mandatory that an application make use of any of these components, it is assumed that most applications will want to use the AcGix library and that a substantial number of them will want to use or adapt the SimpleView code. The WhipView library uses Autodesk's proprietary display list technology to provide a level of "regen-free" panning and zooming.

AcGi

The AcGi API is the interface between AcDb and rendering systems used to display AcDb models. This interface is used by the `AcDbEntity` member functions `worldDraw()`, `viewportDraw()`, and `saveAs()`, which are part of the standard entity protocol.

One method of producing an application capable of basic viewing is to implement fully the AcGi API. Derive your own implementation classes from the AcGi classes, such as `AcGiWorldDraw` and `AcGiWorldGeometry`. To draw a given entity, call its `worldDraw()` function, then pass in a pointer to an instance of your `AcGiWorldDraw`-derived class. You will then receive callbacks into the various members of your class. The member functions are graphics primitives such as the `circle()` and `polyline()` functions. They will be passed all the necessary parameters needed to draw them. AcGi must be implemented by the host application wishing to use specific graphic rendering logic defined by entities. The advantage of using AcGi is that the host application need not know anything about how an entity is intended to be rendered beyond a fixed set of geometric primitives and graphical traits, such as color, linetype, and text font. AutoCAD has its own internal implementation of AcGi, while the AcGix library supplied with ObjectDBX breaks down much of the complex rendering logic specified by AcGi into a relatively simple set of graphics primitives.

Some methods of `AcGiWorldDraw` are for query purposes (`deviation()` and `numberOfIsolines()`) and may be used by an entity to determine the extent

to which various entities will be "tessellated;" in other words, how "dense" the lines making up a sphere (for example) would be.

The `AcGiWorldDraw::regenType()` method can be used to tell AcGi whether the regen request is for wireframes or faces with normals. For example, this is from the *acgi.h* file:

```
// These are the current kinds of viewport regeneration modes.
// This mode cannot be set by the user, but it can be queried
// in case you need to take different actions for different
// regeneration modes.
//
typedef enum {
    eAcGiRegenTypeInvalid = 0,
    kAcGiStandardDisplay = 2,
    kAcGiHideOrShadeCommand,
    kAcGiRenderCommand,
    kAcGiSaveWorldDrawForR12
} AcGiRegenType;
```

NOTE For examples of using the AcGi interface, see the sample module in *samples/common/myacgi.**.

AcGix

This library is an engine that breaks up AcGi-defined geometry and traits into a small, simple set of graphics primitives defined by the protocol of the class `AcGixVectorTaker`. It operates on a registered set of viewports for which the application must provide implementations of `AcGixVectorTaker` and `AcGiViewport`. AcGix queries the supplied `AcGiViewport` for regeneration parameters and translates the AcGi primitives it receives from entities into calls to the supplied vectortaker. AcGix does not make any interpretation of how the application-supplied viewports are actually displayed. This is up to the implementor of `AcGixVectorTaker`.

To use AcGix with a custom graphics system rather than using SimpleView, you must supply your own implementations of the `AcGixVectorTaker` and `AcGiViewport` classes. The actual instances of viewport and vectortaker can be shared between multiple viewports if this makes sense for your application.

It is assumed that the vectortaker implementation will perform the requisite clipping of primitives against the viewport extents. The AcGix library is supplied in binary form with a set of API header files.

The following header files contain the source definition of the AcGix API:

- *include/acgix.h*
- *include/acgixcontext.h*
- *include/acgixstd.h*
- *include/acgixutilities.h*
- *include/acgixvectortaker.h*
- *include/acgixviewportset.h*
- *include/acgixviewportmanager.h*

AcGix clients link with *release/AcGix.lib*, which binds the application to *release/AcGix.dll*.

AcGix Differences from AutoCAD Viewing

The following features are not supported by AcGix:

- Explicit draw order
- XREFs
- Block reference clipping
- Perspective views
- Viewport front/back clipping
- Complex linetypes
- Lineweight
- Plot styles
- TrueType text font elaboration

TrueType Font Elaboration

Although neither SimpleView nor WhipView, through AcGix, supports TrueType fonts, there is a procedure available in ObjectDBX for those who want to try adding TrueType font support on their own. The mechanism is the `textMsg()` method of the `AcGixVectorTaker` class.

When a TrueType text object is to be displayed, AcGix calls the `textMsg()` method of the vectortaker. At this point the vectortaker implementation can either process the text or return `Adesk::kFalse` to instruct AcGix to perform its default processing, which is to convert the text into an SHX font (*txt.shx* by default) and then perform the standard SHX processing.

The coordinates supplied to `textMsg()` are in current model coordinates. These can be converted to the World Coordinate System (WCS) using the supplied transform.

The task of representing TrueType fonts in 3D space is complex, and this exercise is recommended only for those looking to achieve complete

AutoCAD display compatibility. One approach may be only to process this message if the text can be represented in plan view for your viewport, and to otherwise use the default processing.

The `textMsg()` method is defined in *AcGixVectorTaker.h*:

```
virtual Adesk::Boolean
textMsg(
    Adesk::Int16 nViewportId,
    const TextPacket * pPacket) = 0;
```

The `TextPacket` structure contains information about the text plus the transformation matrix used to convert from the current model to WCS.

```
struct TextPacket
{
    TextPacket(
        const TextInfo* pInfo,
        int nColor,
        const AcGeMatrix3d& xModel);
    int m_nColor;
    const TextInfo* m_pInfo;
    const AcGeMatrix3d& m_xCurrentModelToWorld;
};
```

The `TextInfo` structure contains all the information about the text:

```
struct TextInfo
{
    AcGePoint3d m_Position;
    AcGeVector3d m_Normal;
    AcGeVector3d m_Direction;
    double m_Height;
    double m_Width;
    double m_Oblique;
    const char* m_pMsg;
    Adesk::Int32 m_Length;
    Adesk::Boolean m_Raw;
    double m_Thickness;
    const AcGiTextStyle * m_pTextStyle;
};
```

SimpleView

SimpleView is a sample vectortaker. It implements a simple viewport manager and supplies `AcGixSimpleView`. `AcGixSimpleView` aggregates implementations of `AcGiViewport` and `AcGixVectorTaker` into a single object. This implementation uses the Windows GDI to display the results of a regen on the screen. The full implementation of SimpleView is supplied in source form. It can be modified by developers who wish to define a view management system that fits their application's needs. SimpleView is intended to demonstrate what is required to manage a viewport layout and to work with

AcGix to form a complete viewer tool and to serve as a starting point for such an implementation.

SimpleView clients directly link with the library *release/AcGixSimpleView.lib*.

The source and the corresponding Microsoft Visual C++ project file are supplied in the directory *samples/AcGixSimpleView*.

AcGixBlockView provides a base class for different types of views to be managed by the SimpleView manager. This allows both SimpleView and WhipView to be controlled polymorphically by the same manager.

Using the Database Mutex

When an application allows multiple threads to run through the database code, conflicts will eventually arise. These will normally result in errors in `acdbOpenObject` either in your code or in the internal implementations of the API. For this reason, `AcGixSimpleView` maintains a database mutex. Whenever one of the SimpleView threads needs to go into the database code, it tries to obtain this mutex, waiting until it does. ViewAcDb uses the same mutex to access the database. This prevents the two threads from conflicting within the database code.

The basic procedure for multithreaded programming is that whenever you start a database operation, call the `getDatabaseMutex()` method at the start of the operation, and place a call to `releaseDatabaseMutex()` at the end. These calls *must* be paired, otherwise your application will hang. This means that if you have a return statement somewhere in the middle of your code, make sure you have a call to `releaseDatabaseMutex()` before you exit. You must release the mutex as soon as you are done, otherwise the regen thread will wait forever.

You can use the `Mutex` class included in *AcGixSimpleViewManager.cpp* to help you ensure that the calls are paired. The `AcGixSimpleViewManager` class exposes three methods for you to get the mutex. The `databaseMutex()` method is provided for direct access to the mutex. The get and release methods are provided for convenience. The methods are

```
void getDatabaseMutex()
void releaseDatabaseMutex()
HANDLE databaseMutex()
```

WhipView

The WhipView library implements `AcGixView` and `AcGixVectorTaker` on top of the *WHIP!* graphics accelerator. *WHIP!* is a graphics accelerator with a 2D image cache built on top of Autodesk's HEIDI technology. It exports a single API function, `acgixAllocateWhipView()`, which creates and returns an

instance of `AcGixBlockView`. The returned instance can be used in the same way as any other `AcGixBlockView`. The SimpleView library demonstrates the creation of drawing views using WhipView. Because of the display list, WhipView is able to service some actions like pan and zoom without the need for a regen. This gives better performance than SimpleView's direct GDI implementation.

WhipView is supplied in binary form only, and consists of several DLLs and support files representing the WhipView library, the *WHIP!* component, HEIDI, and HDI device drivers. WhipView can be used independent of SimpleView, except for the requisite elements of `AcGixBlockView`. Direct use of the *WHIP!*, HEIDI, and HDI drivers by ObjectDBX developers is not supported. They are supplied in binary form only, with no associated headers.

The API of the WhipView module consists of a single entry point that has the following signature:

```
AcGixBlockView* acgixAllocateWhipView();
```

This function is explicitly declared external and used in the SimpleView source module *AcGixSimpleViewManager.cpp*. There is no exported header file that declares `acgixAllocateWhipView()`.

To reuse this element, you need to take `AcGixBlockView` as defined and as much else of the SimpleView complex that is needed. Be warned, however, that `AcGixBlockView` is rather complicated and uses much of the rest of SimpleView. The implementation is most easily done if you leave SimpleView in an essentially unaltered state.

WhipView clients link directly with *AcGixWhipView.lib*. WhipView requires *AcDb.dll*, *heidi3.dll*, *dllong3.dll,* and the HDI files supplied in the *release* directory.

ViewAcDb

ViewAcDb is a Multiple Document Interface (MDI) drawing file viewer that uses SimpleView and WhipView for displaying views of DWG files. ViewAcDb is essentially a test harness for the *AcDb.dll*, *AcGix.dll,* and the view implementations.

Basic Viewer Operation

The following are the basic steps to establish a view (or views) of a drawing, followed by a description of graphical entity selection.

To establish a view

1. The ObjectDBX application loads the DWG file into an instance of `AcDbDatabase`.
2. The application creates a viewport and passes the associated `AcGiViewport` to AcGix.
3. The application tells AcGix to initiate a regen of a specific block table record or a single entity into a given set of viewports.
4. AcGix regenerates the entity(ies) into all active viewports. Essentially, each entity is opened for read, its `AcDbEntity::worldDraw()` is invoked, and if the return status indicates, its `AcDbEntity::viewportDraw()` member override is invoked once for each active viewport.
5. From either of these members, each class is free to obtain the `AcGiWorld/ViewportGeometry` objects and the `AcGiSubentityTraits` objects, and make calls to send geometric graphics and graphics traits (attributes like color, linetype, font) to them. These AcGi objects are implemented in AcGix, which takes the "input" geometric primitives and traits and processes them, reducing them to the primitives passed in to the instance of `AcGixVectorTaker` associated with each active viewport.
6. The vectortaker takes the input message packets (or function calls) and converts them into calls to the underlying graphics system.

Graphical Entity Selection

The `AcGixVectorTaker::message()` method supplies your application with the information required to implement entity selection (picking). The actual implementation of the selection depends on how you implement your display. For example, WhipView performs selections against its 2D display list.

Configuration Suggestions

An ObjectDBX client application can choose one of the following methods to display the contents of a DWG file:

- Adapt the supplied SimpleView library source for use by the host applications, and use all three components. Study and adapt any code from the ViewAcDb sample application as needed. This is probably the best way to get familiar with the usage of AcGix and of the SimpleView AcGix platform implementation. However, you are free to modify the SimpleView source as desired.
- Build a module to drive a desired graphics system through the AcGix library. This allows much of the existing elaboration logic to be reused and still offers considerable flexibility in actual graphics presentation and performance. This would involve using the AcGix module but writing the implementations of `AcGiViewport` and `AcGixVectorTaker` from scratch, in effect replacing the SimpleView and/or WhipView libraries entirely.
- Use the WhipView subsystem to retain elements of SimpleView needed to support the definition of `AcGixBlockView`. This includes the supplied combination of *WHIP!* and HEIDI DLLs. Direct use of the *WHIP!* and HEIDI components is not supported in this release.
- Write an entire custom implementation of the AcGi interface and do not use any of the supplied AcGix, SimpleView, or WhipView components. You can attain maximum performance this way with a maximum amount of development work.

Demand Loading

The demand loading mechanism is essentially the same for an ObjectDBX application as it is for an ObjectARX application. The only difference is in where the information is found in the registry. For demand loading ObjectARX applications, AutoCAD looks in the system registry under the following:

```
HKEY_LOCAL_MACHINE
    Software
        Autodesk
            AutoCAD
                R15.0
                    ACAD-xxxxxxx-xxxxxxxx
                        Applications
```

`xxxxxxx-xxxxxxxx` is a number unique to each installation.

For demand loading DBX applications, ObjectDBX will look in the system registry under the following:

```
HKEY_LOCAL_MACHINE
    Software
        Autodesk
            ObjectDBX
                R15.0
                    Applications
```

These are hardcoded keys and are the same for any application on any machine. Your application information goes under the `Applications` entry.

Installing the ObjectDBX Libraries

NOTE The following discussion assumes a basic understanding of installer technology, specifically InstallShield 3.0 and 5.0.

The AutoCAD ObjectDBX shared libraries comprise a group of files that were developed with the intent that they be used by multiple applications. This characteristic requires that they be placed in a common location accessible to multiple applications on the user's workstation. Autodesk has chosen Microsoft's recommended implementation of installing files of this nature in the Common Files directory. Since it is possible that your installer may be sharing these files with other applications, you need to follow some simple, yet critical, guidelines when installing the files.

Use COMMONFILES

Do not assume drive letters and paths when determining where to install the Autodesk Shared files. The InstallShield system constant COMMONFILES will return a path name that points to something like *c:\Program Files\Common Files*. You must then append the Autodesk Shared name. This can be done in a *.rul* script with the following line:

```
szSharedPath = COMMONFILES ^ "Autodesk Shared";
```

All files that you are redistributing from the *ObjectDBX\release* directory of your SDK installation should be treated as Autodesk Shared files for installation purposes.

Install by Version and as SHAREDFILE

All DLLs Autodesk provides contain a version resource. When installing these files you should specify that they be updated only if the version is equal to or greater than those that may exist on the user's machine. InstallShield provides a couple of ways to do this using the combination of the `COMP_UPDATE_SAME` and `COMP_UPDATE_VERSION` flags.

In addition you must also mark these files with the `SHAREDFILE` flag when installing. This will ensure that you are maintaining reference counting with other application installers that may be installing and using these libraries and files.

The following InstallShield script is an example that installs the *acge15.dll* using the version and shared file mechanism.

```
TARGETDIR = COMMONFILES ^ "Autodesk Shared";
nReturn = XCopyFile (
    "acge15.dll", "acge15.dll",
    COMP_UPDATE_SAME |
    COMP_UPDATE_VERSION |
    SHAREDFILE);
if  (nReturn < 0 ) then
    // Report failure
endif;
```

Ensure the Files Are on the Path

In order for your application to find these libraries, you must ensure that the user's system path contains a reference to `COMMONFILES ^ "Autodesk Shared"`. There are a number of ways to make this work, depending on the operating system. These are described below.

For All Operating Systems

Update the per-application path on every operating system. Using an InstallShield script, this can be done in the following manner. (`szAppPath` is assumed to include the shared path and any other paths required by your application.)

```
szExe = "your.exe";
szProdKey = "Software\\Microsoft\\Windows\\";
szProdKey = szProdKey + "CurrentVersion\\App Paths\\" + szExe;
RegDBSetKeyValueEx(szProdKey, "Path", REGDB_STRING, szAppPath, -1);
```

Windows NT

You should also update the System Path in the registry. This can be found under `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Environment`. The Path value stored there should be modified to include your path. Note that this string is a `REGDB_STRING_EXPAND` value type.

Windows 95 and Windows 98

You need to update the user's *AUTOEXEC.BAT* file to include the "aliased" (or short) path name because the PATH variable will not handle spaces. While per-application paths should suffice, it is strongly recommended that you update the user's *AUTOEXEC.BAT* due to known problems with automation (refer to Microsoft KnowledgeBase article Q148375).

Ensure Smart Pathing Updates

When updating the user's *AUTOEXEC.BAT* please be diligent about modifying a path entry if the path entry is already there. What should be prevented is just modifying the PATH without regard to either of these conditions:

- The path is already there.
- You are affecting other path settings in the batch file.

When updating the PATH value, no matter which operating system you are dealing with, your installer should prompt the user to reboot so that the path change is properly recorded after installation is complete.

Autodesk provides the InstallShield script below as an incomplete example of smart path updating:

```
function AdUpdateAUTOEXEC (szSharedPath)
STRING szRootPath, szBatchName, szBatchFile,
       szBackupName,szTestLine,szCheckForPathLine;
NUMBER nReturn, nvHandle;
STRING szOutput;
begin
    szOutput = "SET PATH=%PATH%;" + szSharedPath;
    // Obtain the filename of the system batch file.
    BatchGetFileName (szBatchFile);
    ParsePath(szRootPath, szBatchFile, PATH);

    // Make sure we're pointing at the root of the system
    VarSave(SRCTARGETDIR);
    TARGETDIR = szRootPath;
    SRCDIR = szRootPath;
    // See if we have an AUTOEXEC.
    if (Is(FILE_EXISTS,szBatchFile) = FALSE) then
        // If we don't, just write 'ours' out and no more
```

```
        OpenFileMode (FILE_MODE_NORMAL);
        ParsePath(szBatchFile,szBatchFile,FILENAME);
        CreateFile (nvHandle,SRCDIR,szBatchFile);
        WriteLine  (nvHandle, szOutput);
        CloseFile  (nvHandle);
        bNeedReboot = TRUE;
    else
        ParsePath(szBatchName, szBatchFile, FILENAME_ONLY);
        szBackupName = szBatchName + ".ADK";
        ParsePath(szBatchFile, szBatchFile, FILENAME);
        OpenFileMode(FILE_MODE_NORMAL);
        nReturn = OpenFile (nvHandle, SRCDIR, szBatchFile);
        if (nReturn = 0 ) then

            // Spin down to find the non-blank last line in
            // the file
            while (nReturn = 0 )
                nReturn = GetLine(nvHandle, szTestLine);
                if (StrLength(szTestLine) > 0) then
                    szCheckForPathLine = szTestLine;
                endif;
            endwhile;
            CloseFile(nvHandle);

            // We now have the last text entry in the batch
            // file.
            if (StrFind(szCheckForPathLine,szSharedPath) < 0) then
                Disable(LOGGING);
                // Backup up the original
                nReturn = CopyFile (szBatchFile, szBackupName);
                Enable(LOGGING);
                if (nReturn = 0 ) then
                    OpenFileMode (FILE_MODE_APPEND);
                    OpenFile(nvHandle,SRCDIR,szBatchFile);
                    WriteLine(nvHandle,"");
                    WriteLine(nvHandle,szOutput);
                    CloseFile(nvHandle);
                    bNeedReboot = TRUE;
                endif;
            endif;
        endif;
    endif;
    VarRestore(SRCTARGETDIR);
end;
```

Tips and Techniques

Because ObjectDBX operates without AutoCAD, your application must attend to items that an ObjectARX program would not need to address. These items include `AcDbDatabase` and viewport topics usage, utilization of DWG files from earlier releases, and dealing with Extended Entity Data (EED) and raster images. All these topics and more are discussed in this section.

ACAD_OBJID_INLINE_INTERNAL

The header files *acdb.h* and *dbidar.h* contain an `#ifdef` section that selects a header to `#include` based on the `ACAD_OBJID_INLINE_INTERNAL` macro. Applications should never `#define` this value. This `#define` is intended for Autodesk internal use only. Applications that include a `#define ACAD_OBJID_INLINE_INTERNAL` macro will not compile successfully.

AcDbDatabase Notes

ObjectDBX allows you to have several instances of the `AcDbDatabase` class, though you must be sure to delete them all before your application exits. It is also important that you always have one instance of `AcDbDatabase` that is the current database. These requirements are described in the following sections.

Always Instantiate an AcDbDatabase

You must always instantiate at least one `AcDbDatabase` object before using any AcDb code. This is because creating a full `AcDbDatabase` will initialize a set of global variables that some library elements require. (See the `setWorkingDatabase()` method of the `AcDbHostApplicationServices` class in the *ObjectARX Reference* for details.) When AutoCAD is present, there is always a "current drawing" `AcDbDatabase` that ObjectARX relies on to satisfy all these requirements. The ObjectDBX application must emulate this behavior.

If you intend to write directly to your database, you should instantiate it using the `AcDbDatabase` constructor with the `Adesk::kTrue` argument. This version of the constructor creates a viable empty drawing ready for modification and builds the database tables and initializes them to default values. The following is an example of instantiating a database:

```
AcDbDatabase *pDb = new AcDbDatabase(Adesk::kTrue);
```

If you intend to read a drawing file into the database, use the `AcDbDatabase` constructor with the `Adesk::kFalse` argument, immediately followed by a

call to the `readDwgFile()` function. This version of the constructor creates a completely empty database that relies on the subsequent call to `readDwgFile()` to fill out its internal data structures. Using these in combination improves efficiency when reading a DWG file over using the other form of the constructor because the tables and globals need only be initialized once by `readDwgFile()`. Here is an example of reading a drawing into a previously instantiated database:

```
AcDbDatabase *pDb = new AcDbDatabase(Adesk::kFalse);
pDb->readDwgFile(filename);
```

Multiple calls to `readDwgFile()` on the same database are not supported.

In both ObjectARX and ObjectDBX, calling `readDwgFile()` after using the `Adesk::kTrue` form of the constructor is certain to cause failure if the version of the DWG file you are reading is Release 12 or earlier. This is due to an inconsistency in the way drawings are loaded prior to AutoCAD Release 13. Because you cannot predict which drawings your end-users will open, do not code the following:

```
// Do not do this.
AcDbDatabase *pDb = new AcDbDatabase(Adesk::kTrue);
pDb->readDwgFile(filename);
```

Always Have a Current Database

Although ObjectDBX allows you to create many instances of `AcDbDatabase`, only one may be the current database. AcDb maintains an internal global pointer to this current database and the `AcDbHostApplicationServices` method `workingDatabase()` returns a copy of this internal pointer. Whenever the `AcDbDatabase::readDwgFile()` method is used, the internal current database pointer is reset to point to that database. When the database that is current is deleted, the internal current database pointer is set to `NULL`.

This means that in situations where your code reads multiple databases and perhaps deletes some of them along the way, you must be sure that you use the `AcDbHostApplicationServices::setWorkingDatabase()` method in appropriate places to set the internal current database pointer to the proper current database.

For example, the following code reads in one database pointed to by `pDbFirst` (call this database "first"). This causes the internal current database pointer to be set to database "first." The code then reads in another database pointed to by `pDbSecond` (call this database "second"), which now causes the internal current database pointer to be set to database "second." The code next inserts database "second" into database "first" and deletes database "second." When database "second" is deleted, the internal current database pointer is pointing to it, so the internal pointer is set to `NULL`. This means that

after the database deletion, AcDb has a `NULL` current drawing pointer. This leads to fatal errors if any code that references the internal current database pointer is accessed.

To prevent this, after the deletion of database "second" in your application's code, the code needs to call the host service's `setWorkingDatabase()` method, passing in a pointer to database "first" to reestablish database "first" as the current database for ObjectDBX, as follows:

```
// Make "first" the current database.
AcDbDatabase *pDbFirst = new AcDbDatabase(Adesk::kFalse);
pDbFirst->readDwgFile("first.dwg");

// Now make "second" be the current database.
AcDbDatabase *pDbSecond = new AcDbDatabase(Adesk::kFalse);
pDbSecond->readDwgFile("second.dwg");

// Insert "second" into "first" as ABLOCK.
Acad::ErrorStatus es;
AcDbObjectId blockId;
es = pDbFirst->insert(blockId, "ABLOCK", pDbSecond);

// Deleting "second" makes the current database NULL.
delete pDbSecond;

// Make the current database "first" again.
myHostServices->setWorkingDatabase(pDbFirst);
```

Delete All AcDbDatabases at Application Exit

Always delete all databases before exiting your application. Aside from causing a memory leak in your application, failure to delete all databases can result in a fatal error on shutdown. Please also see the following section, "AcDbDatabase::insert()," regarding special destruction considerations for inserted databases.

AcDbDatabase::insert()

When inserting one database into another, the order of destruction is critical. When using this function, the database executing the insert is the "To" database, and the database used as an argument is the "From" database. Always destroy the "From" database first, and the "To" database last; otherwise you will cause a fatal error in the ObjectDBX DLL.

Finding the Active Viewports in Model Space

When creating a new DWG file with the `AcDbDatabase(Adesk::kTrue)` constructor, you must set the default model space viewport to some reasonable parameters. This will ensure that the DWG will be visible in AutoCAD when opened. Failure to set the default model space viewport may require the

AutoCAD user to select Zoom All to make the geometry of the drawing visible. Furthermore, some AutoCAD entities require reasonable viewport parameters to calculate aspects of their appearance. For example, an `AcDbSpline` object created in ObjectDBX, and saved to a drawing without any attention to the viewport, may display with sharp angles, like a pline, when loaded into AutoCAD. The data is not damaged and the first time the entity is edited in AutoCAD the `AcDbSpline` will revert to its proper shape. However, this is certainly disconcerting to users of your product's drawings. Developers possessing an asserts-enabled AutoCAD are likely to see asserts fire when loading a drawing that was saved without proper attention to the viewport. Experimentation will demonstrate which parameters are best for your application and geometry. To set the model space viewport, insert the following:

```
// Set some viewport information.
AcDbViewportTable* pViewportTable;
if (db.getViewportTable(pViewportTable, AcDb::kForRead)
    == Acad::eOk)

{
    // Find the first viewport and open it for write.
    AcDbViewportTableRecord *pRecord;
    if (pViewportTable->getAt(
        "*ACTIVE", pRecord,
          AcDb::kForWrite) == Acad::eOk)
    {
        pRecord->setCenterPoint(AcGePoint2d(0.5, 0.5));
        pRecord->setHeight(1.0);
        pRecord->setWidth(1.0);
        pRecord->close();
    }
    pViewportTable->close();
}
```

Details About Viewports

There are a few subtleties regarding the various viewport types in AcDb that are worth review.

An `AcDbDatabase` object can contain both model space and paper space viewports. These are represented by different AcDb types internally.

Model space viewports are represented by `AcDbViewportTableRecords`, which are contained in the `AcDbViewportTable`. These are completely unrelated to `AcDbViewport` entities. AutoCAD Release 13 and higher require the existence of at least one `AcDbViewportTableRecord` in the `AcDbViewportTable`. It will be named "*ACTIVE," which implies that it was the active viewport when the drawing was saved. More than one viewport

can be "*ACTIVE" simultaneously. It is illegal to have a `ViewportTableRecord` without a name.

Paper space viewports are represented by `AcDbViewport`. These may only exist within the paper space block table record of the `AcDbBlockTable`. Paper space itself must have the main paper space viewport as an `AcDbViewport` entity in the paper space block table record. In AutoCAD, this default viewport is created automatically when the TILEMODE 0 command is executed. There is an assumption in the API that this viewport will be created automatically. Thus, when a new `AcDbViewport` is instantiated and added to the paper space block table record, if the main paper space viewport does not yet exist, it will be created during the close of the new `AcDbViewport`. Note that this means it is never necessary to create the main paper space viewport explicitly. It is perfectly valid to add entities to paper space without ever creating a paper space viewport. This is because AutoCAD can successfully open a drawing of this nature, and will automatically create the main paper space viewport the first time the TILEMODE 0 command is used. AutoCAD should correctly display the entities in the paper space block table record within the main paper space viewport at that time.

In ObjectDBX, the `AcDbViewport::number()` function will always return –1. In ObjectARX, it reports the viewport number of the current viewport in the AutoCAD editor. Because AutoCAD is not present in ObjectDBX, this value has no meaning. ObjectDBX does provide the `acdbGetCurVportId()` function, which returns the current object ID of the viewport when the drawing was saved.

It is highly recommended that you review the ObjectARX SDK documentation regarding viewports of all types.

Always Test Your Drawings in AutoCAD 2000

During development, ObjectDBX application developers should frequently check the validity of their implementation by opening drawing files saved by their application in AutoCAD 2000. This will also aid you in determining how to best set your model space viewport.

Using DWG Files from Earlier Releases

When ObjectDBX opens any drawing file, it converts the file to the AutoCAD 2000 format. ObjectDBX can read them in and save them back out as Release 13, Release 14, or AutoCAD 2000, and the data won't change in any way. This means that an original drawing and a copy created with ObjectDBX look the same when opened in AutoCAD.

For Release 12 and earlier drawings, however, differences in the drawing format make the conversion much more extensive. Because of this, when a Release 12 or earlier DWG file is converted and then saved by ObjectDBX, it may appear off-center the first time it is opened in AutoCAD. The reason for this is that part of the conversion process AutoCAD uses involves establishing a view based on the windowing situation. ObjectDBX doesn't have windows, and so cannot establish the same view. Performing a Zoom All or Zoom Extents from the Zoom submenu of the AutoCAD View menu will recenter the drawing.

When evaluating the performance of ObjectDBX, please keep in mind that reading AutoCAD Release 14 and earlier drawings causes a conversion to take place, which affects the open time.

Extended Entity Data

Registering Extended Entity Data (EED) is supported in ObjectDBX. A few minor differences from the ObjectARX API exist, however. To use EED, an application must be registered with the database. In the ObjectARX API, the `acdbRegApp()` function would most likely be utilized. This interface is not supported in ObjectDBX, so alternative functions for registering an application and dealing with resbuf chains are described here.

To register your application, use the `AcDbDatabase` functions to get the `regapp` table, and add a record:

```
registerApplication(AcDbDatabase* pDatabase)
{
    AcDbRegAppTable *pRegAppTable;
    AcDbObjectId blockId;
    if (pDatabase->getRegAppTable(pRegAppTable, AcDb::kForWrite)
        == Acad::eOk)

    {
        AcDbRegAppTableRecord *pRecord = new AcDbRegAppTableRecord;
        if (pRecord)
        {
            pRecord->setName("ACDBTEST_APP"); // For example
            if (pRegAppTable->add(blockId, pRecord) == Acad::eOk)
                 pRecord->close();
            else delete pRecord;
        }
        pRegAppTable->close();
    }
}
```

EED is added to an `AcDbEntity` as a resbuf chain. When using resbuf types that require pointers (like `resval.rstring`), be sure to allocate the pointer with the `acdbAlloc()` function, and delete it with the `acdbFree()` function (declared in the *dbmain.h* file).

Raster Images

If you wish to write an ObjectDBX application that manipulates raster entities, you must first link to the Imaging Support Module (ISM) DBX, then have your application explicitly load that DBX file before attempting to call any of the raster APIs. For example, if you were using the AutoCAD LT level of ACIS support, include this call:

```
AcRxDynamicLinker->loadModule( "acIsmobj.dbx" );
```

If you read a DWG file that contains raster entities, ObjectDBX will attempt to load *acIsmObj.dbx* upon encountering an `AcDbRasterImage` entity in the drawing file. ObjectDBX will only search the saved path for the image file. This differs from AutoCAD, which in addition searches the AutoCAD search path.

Remember that the Image Engine readers (*ie*rd.dll*, in this case) must be in the same directory as the *ism*.dbx* files. It is not sufficient for these files to be on the search path; instead they should be in the same location as the *ism*.dbx*. This behavior matches that of AutoCAD.

Known Limitations

Please review the "Tips" section for the `AcDbDimension` class in the *ObjectARX Reference*, which documents the behavior of `AcDbDimension` entities in an external database. For the purposes of modifying or creating new `AcDbDimension` entities by the API, every ObjectDBX database behaves as an external database. Thus, a newly created or modified `AcDbDimension` object will have its `dimBlockId` set to `NULL`. Calling the `acdbMakeDatabaseCurrent()` function is not sufficient to change the behavior documented in the ObjectARX *Readme*. This does not prevent the creation of a valid drawing, as AutoCAD is capable of generating the correct `dimBlockID` for an `AcDbDimension` at regen time.

The Graphics Interface Library

AutoCAD uses the graphics interface library (AcGi) to display built-in and custom entities. This chapter discusses setting entity traits and using primitives to create custom graphical entities. For a complete description of all AcGi classes and their member functions, see the *ObjectARX Reference*.

In This Chapter

26

- AcGi Overview
- Setting Entity Traits
- Primitives
- Using Drawables in Your Object
- Tessellation
- Isolines
- Transformations
- Using Clip Boundaries in AcGi

AcGi Overview

The AcGi library defines a set of interfaces with which objects can render themselves to an underlying graphics system. This chapter discusses how AcGi works in the AutoCAD environment. However, it works in a similar way for other systems that implement the AcGi interfaces.

The AcGi library enables entities to query for information about the regeneration process, and to detail a set of primitives using the geometry classes. Access to AcGi occurs within the following three member functions of the `AcGiDrawable` base class:

```
Adesk::Boolean
worldDraw(
    AcGiWorldDraw*);

void
viewportDraw(
    AcGiViewportDraw*);

Adesk::UInt32
setAttributes(
    AcGiDrawableTraits*);
```

`AcDbEntity` inherits these functions from `AcGiDrawable`. Typically, when implementing a custom entity, you will override these functions and provide your own implementation.

When AutoCAD needs to regenerate the graphics to display an entity, it calls these functions in the following manner:

```
AcGiDrawable *pDrawable;
pDrawable->setAttributes(pDt);
if (!pDrawable->worldDraw(pWd))
{
    for each viewport
        pDrawable->viewportDraw(pVd);
}
```

For custom entities, AutoCAD calls your `setAttributes()`, `worldDraw()`, and `viewportDraw()` functions if you have overridden them. AutoCAD passes in the appropriate AcGi objects to these functions. This enables AutoCAD to display your custom entity just as if it were a built-in entity.

The `setAttributes()` function initializes attributes for the entity, such as color, layer, and linetype. The `worldDraw()` function builds the portion of the entity's graphical representation that can be specified independent of any particular model-space view or paper-space viewport contexts. The `viewportDraw()` function then builds the view-dependent portion of the entity's graphics. If any of the entity's graphics are view-dependent,

`worldDraw()` must return `kFalse` and `viewportDraw()` must be implemented. Conversely, if the entity has no view-dependent graphics, then `worldDraw()` must return `kTrue` and the custom entity does not implement `viewportDraw()`.

The following illustration shows the sequence in which an AutoCAD drawing with two viewports gets regenerated. In this example the drawing contains two blocks, Block 1 and Block 2. Block 1 is broken down into its component parts, a line and a circle. Block 2 consists of a custom entity. The custom entity is broken down to show the order in which functions are called as the drawing is generated:

The `AcGiContext` object provides a common context that can be accessed during all parts of the regeneration process. It provides information about the current state of the regen. For example, you can get the current database from the `AcGiContext` object at any time during the regen process.

The class hierarchy for AcGi is as follows:

```
AcRxObject
    ├── AcGiCommonDraw
    │    ├── AcGiWorldDraw
    │    └── AcGiWorldDraw
    ├── AcGiContext
    ├── AcGiEdgeData
    ├── AcGiFaceData
    ├── AcGiGeometry
    │    ├── AcGiViewportGeometry
    │    └── AcGiWorldGeometry
    ├── AcGiLinetypeEngine
    ├── AcGiSubEntityTraits
    │    └── AcGiDrawableTraits
    ├── AcGiTextStyle
    ├── AcGiVertexData
    ├── AcGiViewport
    └── AcGiDrawable
         └── AcGiGlyph
```

The `AcGiCommonDraw` base class encapsulates the common functionality of `AcGiViewportDraw` and `AcGiWorldDraw`. The `AcGiGeometry` base class encapsulates the common functionality of `AcGiViewportGeometry` and `AcGiWorldGeometry`. These base classes allow you to write more general code that can handle both cases, if desired.

The setAttributes Function

The AcGi implementation calls the `setAttributes()` function on a drawable to obtain the general rendering attributes to be used for the object's geometry, such as color, layer, and linetype. This call is made prior to calling the object's `worldDraw()` function, and AcGi uses the return value to determine how to handle most efficiently the object later on.

Normally you will not need to override the `setAttributes()` function, but if you do so, you must call the base class implementation first and return the same flags in the return value. The only alteration that can be made is to add additional flags to the return value. For more information on the return flags, see `AcGiDrawable::setAttributes()` in the *ObjectARX Reference*.

The default implementation of `AcDbEntity::setAttributes()` sets up the color, layer, linetype, lineweight and plot style using the entity's current properties, and the return value includes the `kDrawableIsEntity` flag.

The worldDraw() Function

The `worldDraw()` function is the primary mechanism for a drawable to display itself. From this callback the drawable uses the `AcGiSubEntityTraits` and `AcGiWorldGeometry` interfaces to tell the AcGi implementation how this drawable should be represented in all active viewports. For information that is specific to certain viewports, the `viewportDraw()` callback is provided (see "The viewportDraw() Function" on page 688).

The `AcDbEntity::worldDraw()` function takes a pointer to an `AcGiWorldDraw` object. `AcGiWorldDraw` is a container class for the AcGi geometry and traits objects. Specifically, `AcGiWorldDraw` contains two other objects:

- `AcGiWorldGeometry`
- `AcGiSubEntityTraits`

The `AcGiWorldGeometry` object can be accessed from within `worldDraw()` by using the `AcGiWorldDraw::geometry()` function, and the `AcGiSubEntityTraits` object can be accessed by using the `AcGiWorldDraw::subEntityTraits()` function.

The `AcGiWorldGeometry` object writes vectors to AutoCAD's display using its set of drawing primitives. A *primitive* is the lowest-level instruction used to draw graphical entities. The world geometry object has the following functions for drawing primitives in world coordinates, which are inherited from `AcGiGeometry`:

- Circle
- Circular arc
- Polyline
- Polygon
- Mesh
- Shell
- Text
- Xline
- Ray
- Draw

The draw method allows you to specify another drawable to be used as a part of your geometry. This might be another entity or an in-memory drawable. AcGi uses the same `setAttributes()`, `worldDraw()`, and `viewportDraw()` logic on this object as it uses on your object.

The `AcGiSubEntityTraits` object sets graphical attribute values using its set of traits functions:

- Color
- Layer
- Linetype
- Polygon fill type
- Selection marker
- Line weight
- Thickness
- Plot style name (should not be modified during `worldDraw()` or `viewportDraw()`)

The viewportDraw() Function

If the drawable returns false from the `worldDraw()` callback, it invokes `viewportDraw()` once for each active viewport to allow the drawable to describe its viewport-specific display. `AcGiViewportDraw` and `AcGiViewportGeometry` are closely related to the `worldDraw()` equivalents, because they derive from the same base classes.

The `viewportDraw()` function works in a similar way as `worldDraw()`. The `AcDbEntity::viewportDraw()` function takes a pointer to an `AcGiViewportDraw` object and builds the view-specific representation of an entity. The viewport draw object also provides access to other objects, which include the following:

- `AcGiViewportGeometry`
- `AcGiSubEntityTraits`
- `AcGiViewport`

The viewport geometry object provides the same list of primitives as the world geometry object and adds to it the following primitives, which use eye- and display-space coordinates to draw polylines and polygons:

- `polylineEye()`
- `polygonEye()`
- `polylineDc()`
- `polygonDc()`

The viewport subentity traits object is the same as that used by the world draw object (`AcGiSubEntityTraits`). The viewport object provides functions for querying the viewport's transformation matrices and viewing parameters.

WARNING! A pointer to an AcGi object such as `AcGiWorldDraw` or `AcGiViewportDraw` should not be stored as a global or static variable. Do not save copies of pointers to AcGi objects across calls to `worldDraw()` and `viewportDraw()`. Once these functions return, the pointers are no longer valid.

Viewport Regeneration Type

Since AcGi is just an interface specification, it can be used for many different purposes. In AutoCAD the AcGi specification has several different implementations, each with a specific purpose. The 2D-display pipe is used to generate the 2D images displayed on the screen, and there is a different 3D system for the persistently rendered views. In addition to these display-related implementations, there are several others, including any that you may develop.

It is important to be aware of two other AcGi implementations used by the database engine. The first is for proxy graphics generation and the second is used to explode an object into basic ObjectDBX entities. The proxy graphics implementation is used when a custom object is saved, to store the graphics in a metafile that can be replayed when the drawing is loaded on a system where the custom application is not available. The explode implementation is used by the explode mechanism and to discover boundaries within a complex object.

It might be important for you to distinguish between these different implementations. Therefore, AcGi exposes a viewport regeneration type that can be queried from your `worldDraw()` method.

The viewport regeneration type, referred to as the regen type, is set by AutoCAD. You can query its value using the `AcGiWorldDraw::regenType()` function. Values for `AcGiRegenType` are

- `kAcGiStandardDisplay` is the typical drawing mode and is used when the user issues a REGEN command or edits an entry. Entities should be rendered in wireframe in this mode.
- `kAcGiHideOrShadeCommand` performs hidden line removal and indicates that the HIDE or SHADE command is in effect. Entities should be rendered using faces in this mode.
- `kAcGiRenderCommand` uses materials and lighting models to create a realistically shaded image of a 3D model and is used when the user issues a RENDER command. Entities should be rendered using faces in this mode.

- `kAcGiSaveWorldDrawForR12` is the type used for an explode operation.
- `kAcGiSaveWorldDrawForProxy` is the type used for the generation of proxy graphics. In this case all of your rendering should be done in `worldDraw()` since `viewportDraw()` is not supported for proxy graphics.

Setting Entity Traits

There are three levels from which entity color, layer, and linetype values can be set: drawable level, subentity level, and subprimitive level. Other traits can be set from only one or two of the levels:

- *Drawable Level* The implementation of `setAttributes()` specifies the default traits for the primitives used to display the drawable. For most entities, the entire object is rendered using the entity's current properties: linetype, color, layer, and so on.
- *Subentity Level* You can specify specific traits to be used for specific parts of the drawable during the `worldDraw()` or `viewportDraw()` implementation. You can use the `AcGiSubEntityTraits` interface to override traits that were specified in the `setAttributes()` call. Once a value for a trait is set it is used for all subsequent primitives until the end of the method or until a new value is specified.

NOTE In this chapter, the term *subentity* is used differently than in chapter 6, "Entities," where the term refers to specific geometric pieces of an entity. In this chapter, subentity is not a piece of an entity; it is just a level at which trait values can be set and changed.

- *Subprimitive Level* The mesh and shell primitive functions have optional parameters that let you specify a rich set of traits on a per-edge and per-face basis. (See the code samples in "Primitives" on page 696.) For any trait, this mechanism requires that you set values for all of the edges or faces, or for none of them. You set only the traits you want. For example, you can set the colors of the edges of a shell or mesh without having to set layers or linetypes, but you must specify a color for every edge. In addition to mesh and shell subprimitive traits, there is a version of the text primitive function that has a text style parameter. Text style can be set only at the subprimitive (per-text primitive) level. Subprimitive trait values supersede values of the corresponding traits set at the subentity and drawable levels.

Subentity Traits

The following traits (properties) can be assigned at the subentity level by calling member functions of the `AcGiSubEntityTraits` object:

- Color
- Layer
- Linetype
- Fill type
- GS marker
- Line weight
- Thickness
- Line type scale

Color, layer, and linetype are AutoCAD entity properties, so they can also be set at the drawable level as described in the previous section. Fill type and GS marker are not AutoCAD entity properties.

Before each call to `worldDraw()` and `viewportDraw()`, AutoCAD calls `setAttributes()` to allow the drawable to initialize the color, layer, linetype, line weight, thickness, and line type scale subentity traits. It initializes fill type to correspond to the regen type, and it initializes the GS marker to zero (a zero marker signifies "no marker").

Fill Type

The fill type enumerated value, `AcGiFillType`, can have one of two values:

- `kAcGiFillAlways`
- `kAcGiFillNever`

Primitives that can be filled are circles, polygons, shells, meshes, text, arc sectors, and arc chords. Polylines and simple arcs cannot be filled.

Before AutoCAD calls `worldDraw()`, it sets the fill type depending on the regen type. If the regen type is `kAcGiStandardDisplay`, AutoCAD sets the fill type to `kAcGiFillNever`. Otherwise, AutoCAD sets the fill type to `kAcGiFillAlways`. This value is reinitialized according to the regen type before `viewportDraw()` is called.

If the user issues a FILL command specifying to turn Fill mode off, no objects are filled regardless of the regen type. Similarly, if the user explicitly turns Fill mode on, objects will be filled. If the user does not issue a FILL command, and `AcGiSubEntityTraits::setFillType()` has been set, that Fill mode is used regardless of the regen type.

GS Markers

GS markers are mainly useful in the object snap implementation of an entity. When the entity is selected for object snap, the GS marker for the selected portion of the entity is passed back to indicate which points should be returned.

GS markers are also used in conjunction with the functions `acedSSGet()` and `acedSSNameX()` to permit your application to edit or operate on arbitrary sections of your custom entity objects. For a detailed description of how to use GS markers (not how to set them), including the use of the `acedSSGet()`, `acedSSNameX()`, and `AcDbEntity::getSubentPathsAtGsMarker()` functions, see "GS Markers and Subentities" on page 111.

The examples in chapter 6, "Entities," set a GS marker for every edge of the entity. Your custom entity can use markers to identify a set of arbitrary sections of the entity—that is, any sequentially executed group of primitives can be identified by a single marker. The section of the entity generated by the group of primitive function calls is identified by preceding the primitives with a call to the `AcGiSubEntityTraits` function `setSelectionMarker()`, specifying a marker number unique to the entity object. Your implementation of `getSubentPathsAtGsMarker()` will associate the appropriate primitives with a given marker, based on how you set your markers.

Useful AcGi Constants

The following constants are useful when you are setting or querying entity properties:

```
// Color
//
static const Adesk::UInt16 kColorByBlock = 0;
static const Adesk::UInt16 kRed     = 1;
static const Adesk::UInt16 kYellow  = 2;
static const Adesk::UInt16 kGreen   = 3;
static const Adesk::UInt16 kCyan    = 4;
static const Adesk::UInt16 kBlue    = 5;
static const Adesk::UInt16 kMagenta = 6;
static const Adesk::UInt16 kWhite   = 7;
static const Adesk::UInt16 kColorByLayer = 256;

// Linetype
//
static const char* const kNoLinetyping = "CONTINUOUS";
static const char* const kLinetypeByLayer = "BYLAYER";
static const char* const kLinetypeByBlock = "BYBLOCK";

// Layer
//
static const char* const kLayerZero = "0";
```

NOTE Constant `kWhite` is white unless it conflicts with the background color, in which case it becomes black so that it remains visible. If you assign color by block (`setColor(0)`) or color by layer (`setColor(256)`), you'll need to query the block or layer for the actual color value.

Example of Using AcGi

The following example illustrates how to use AcGi. First, it saves the current entity property values for color, linetype, and layer. Then it changes the subentity trait color value to blue and draws a three-point polyline. Next, it changes the subentity trait color value to the current layer color, changes the linetype value to `DASHDOT`, and draws an xline.

The sample code includes two functions, `getLinetypeFromString()` and `getLayerIdFromString()`, which allow you to obtain an ID for a linetype or layer from a string.

NOTE In practice, these functions might be too slow to use within `worldDraw()`, and object IDs should be stored and used directly.

```
static Acad::ErrorStatus
getLinetypeIdFromString(const char* str, AcDbObjectId& id);
static Acad::ErrorStatus
getLayerIdFromString(const char* str, AcDbObjectId& id);

Adesk::Boolean
AsdkTraitsSamp::worldDraw(AcGiWorldDraw* pW)
{
    // At this point, the current property traits are
    // the entity's property traits.  If the current
    // property traits are changed and you want to
    // reapply the entity's property traits, this is
    // the place to save them.
    //
    Adesk::UInt16 entity_color
        = pW->subEntityTraits().color();
    AcDbObjectId  entity_linetype
        = pW->subEntityTraits().lineTypeId();
    AcDbObjectId  entity_layer
        = pW->subEntityTraits().layerId();

    // Override the current color and make it blue.
    //
    pW->subEntityTraits().setColor(kBlue);

    // Draw a blue 3-point polyline.
    //
    int num_pts = 3;
```

```
    AcGePoint3d *pVerts = new AcGePoint3d[num_pts];
    pVerts[0] = AcGePoint3d(0.0, 0.0, 0);
    pVerts[1] = AcGePoint3d(1.0, 0.0, 0);
    pVerts[2] = AcGePoint3d(1.0, 1.0, 0);

    pW->geometry().polyline(num_pts, pVerts);

    // Force the current color to use current layer's color.
    //
    pW->subEntityTraits().setColor(kColorByLayer);

    // Force current line type to DASHDOT.  If
    // DASHDOT is not loaded, the current linetype
    // will still be in effect.
    //
    AcDbObjectId dashdotId;
    if (getLinetypeIdFromString("DASHDOT", dashdotId)
        == Acad::eOk)
    {
        pW->subEntityTraits().setLineType(dashdotId);
    }

    // Force current layer to "MY_LAYER".  If
    // "MY_LAYER" is not loaded, the current layer
    // will still be in effect.
    //
    AcDbObjectId layerId;
    if (getLayerIdFromString("MY_LAYER", layerId)
        == Acad::eOk)
    {
        pW->subEntityTraits().setLayer(layerId);
    }

    // Draw a DASHDOT xline in "MY_LAYER"'s color.
    //
    pW->geometry().xline(pVerts[0], pVerts[2]);
    delete [] pVerts;
    return Adesk::kTrue;
}

// A useful function that gets the linetype ID from the
// linetype's name; the name must be in upper case.
//
static Acad::ErrorStatus
getLinetypeIdFromString(const char* str, AcDbObjectId& id)
{
    Acad::ErrorStatus err;

    // Get the table of currently loaded linetypes.
    //
    AcDbLinetypeTable *pLinetypeTable;

    err = acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pLinetypeTable, AcDb::kForRead);
    if (err != Acad::eOk)
        return err;
```

```
    // Get the ID of the linetype with the name that
    // str contains.
    //
    err = pLinetypeTable->getAt(str, id, Adesk::kTrue);

    pLinetypeTable->close();

    return err;
}

// A useful function that gets the layer ID from the
// layer's name; the layer name must be in upper case.
//
static Acad::ErrorStatus
getLayerIdFromString(const char* str, AcDbObjectId& id)
{
    Acad::ErrorStatus err;

    // Get the table of currently loaded layers.
    //
    AcDbLayerTable *pLayerTable;

    err = acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pLayerTable, AcDb::kForRead);
    if (err != Acad::eOk)
        return err;

    // Get the ID of the layer with the name that str
    // contains.
    //
    err = pLayerTable->getAt(str, id, Adesk::kTrue);

    pLayerTable->close();

    return err;
}
```

Primitives

With mesh and shell primitives, you can specify traits for edges, faces, or vertices in addition to the basic geometry. The following sections illustrate the use of these primitives.

Mesh

A mesh is an efficient way to store a parametrically rectangular grid of vertices. The geometry for a mesh is specified as the number of rows, the number of columns, and a list of vertices, in row-order:

```
virtual Adesk::Boolean
AcGiWorldGeometry::mesh(
    const Adesk::UInt32 rows,
    const Adesk::UInt32 columns,
    const AcGePoint3d*  pVertexList,
    const AcGiEdgeData* pEdgeData = NULL,
    const AcGiFaceData* pFaceData = NULL,
    const AcGiVertexData* pVertexData = NULL) const = 0;
```

The `mesh()` function has three optional parameters for attaching property data to edges, faces, or vertices. For edges in the mesh, you can attach color, layer, linetype, GS marker, and visibility properties. For example, you could use `AcGiEdgeData::setColors()` to attach a different color to each edge of the mesh. In the color list, first list the colors for all the row edges, then the colors for all the column edges. The following figure shows the ordering of edge property data for a sample mesh:

The following sample code creates a mesh and assigns colors using edge data and face data. It constructs a four-by-four mesh with cyan rows and green columns.

```
Adesk::Boolean
AsdkMeshSamp::worldDraw(AcGiWorldDraw* pW)
{
    Adesk::UInt32 i, j, k;
    Adesk::UInt32 numRowS = 4;
    Adesk::UInt32 numCols = 4;
    AcGePoint3d *pVerts =
        new AcGePoint3d[numRows * numCols];

    for (k = 0, i = 0; i < numRows; i++) {
        for (j = 0; j < numCols; j++, k++) {
            pVerts[k].x = (double)j;
            pVerts[k].y = (double)i;
            pVerts[k].z = 0.;
        }
    }

    // Construct an array of colors to be applied to each
    // edge of the mesh.  In this example, the rows are cyan and
    // the columns are green.
    //
    AcGiEdgeData edgeInfo;
    Adesk::UInt32 numRowEdges = numRows * (numCols - 1);
    Adesk::UInt32 numColEdges = (numRows - 1) * numCols;
    Adesk::UInt32 numEdges = numRowEdges + numColEdges;
    short *pEdgeColorArray = new short[numEdges];

    for (i = 0; i < numEdges; i++) {
        pEdgeColorArray[i] =
            i < numRowEdges ? kCyan : kGreen;
    }
    edgeInfo.setColors(pEdgeColorArray);

    // Make the first face transparent and the rest
    // different colors.
    //
    Adesk::UInt32 numFaces = (numRows - 1)
        * (numCols - 1);
    Adesk::UInt8 *pFaceVisArray =
        new Adesk::UInt8[numFaces];
    short *pFaceColorArray = new short[numFaces];
    AcGiFaceData faceInfo;
    faceInfo.setVisibility(pFaceVisArray);

    for (i = 0; i < numFaces; i++) {
        pFaceVisArray[i] =
            i ? kAcGiVisible : kAcGiInvisible;
        pFaceColorArray[i] = (short)(i + 1);
    }

    faceInfo.setColors(pFaceColorArray);
```

```
    // If the fill type is kAcGiFillAlways, then a shell,
    // mesh, or polygon will be interpreted as faces;
    // otherwise, they will be interpreted as edges.
    //
    // Output mesh as faces.
    //
    pW->subEntityTraits().setFillType(kAcGiFillAlways);
    pW->geometry().mesh(numRows, numCols, pVerts, NULL,
        &faceInfo);

    // Output mesh as edges over the faces.
    //
    pW->subEntityTraits().setFillType(kAcGiFillNever);
    pW->geometry().mesh(numRows, numCols, pVerts,
        &edgeInfo);
    delete [] pVerts;
    delete [] pEdgeColorArray;
    delete [] pFaceColorArray;
    delete [] pFaceVisArray;

    return Adesk::kTrue;
}
```

For faces in a mesh, you can attach color, layer, GS marker, normal, and visibility traits. To assign properties to faces in a mesh, you list the values for the faces in row-order, as indicated by the following figure:

0	1	2
3	4	5
6	7	8

Vertex data for the mesh is listed in the same order as in the vertex list. Properties that can be set with `AcGiVertexData` are normals and orientation.

Visibility

The `AcGiEdgeData` and `AcGiFaceData` classes allow you to specify the visibility type for the edges or faces in a mesh or shell primitive. There must be exactly one visibility entry in the array for each edge or face in the primitive. Passing in an array of an incorrect size causes unpredictable results.

The visibility type for edges and faces, `AcGiVisibility`, can have one of the following values:

- `kAcGiInvisible`
- `kAcGiVisible`
- `kAcGiSilhouette`

If the surface is not curved, or the edge is not required for viewing purposes, specify `kAcGiInvisible`. For hard edges of a surface or visible creases, specify `kAcGiVisible`. For edges or faces that you can see from certain viewpoints, specify `kAcGiSilhouette`. The silhouette visibility type is recognized only by the HIDE command; otherwise, it is interpreted as `kAcGiVisible`.

For example, in the solid cylinder shown below, the edges that form the rims of the cylinder are visible edges. The latitudinal edges are invisible edges, since they are never used for viewing purposes. The longitudinal edges are silhouette edges, since they are used when the cylinder is viewed from certain angles.

Shell

A shell is a list of faces that might be connected and can have holes in them. The shell is specified by the number of unique vertices, a list of vertices (`pVertexList`), the number of faces (`faceListSize`), and a face list, which consists of the number of points in a given face followed by the index in the vertex list of each vertex for that face. The signature for the `shell()` function is

```
virtual Adesk::Boolean
AcGiWorldGeometry::shell(
    const Adesk::UInt32   nbVertex,
    const AcGePoint3d*    pVertexList,
    const Adesk::UInt32   faceListSize,
    const Adesk::Int32*   pFaceList,
    const AcGiEdgeData*   pEdgeData = NULL,
    const AcGiFaceData*   pFaceData = NULL,
    const AcGiVertexData* pVertexData = NULL
    const struct resbuf*pResBuf = NULL) const = 0;
```

A negative vertex count indicates a hole in the shell. Holes must be in the same plane as the face in which they reside. The holes must not touch each other and must be completely inside the containing face. The `shell()` function is a costly operation because it requires the use of a triangulator to break the containing face and the holes down into component triangles.

AcGi polygons and shells with faces of five or more sides are also broken down into triangles before being sent to be displayed. Having the AcGi triangulate a polygon or shell face can be costly in terms of memory and speed, so it's recommended you use three- or four-sided faces in shells to build up faces or polygons with five or more sides. That way, the primitive will not be put through the slow triangulator step.

NOTE The triangulator is used only on polygons of five sides or more, shell faces of five sides or more, shell faces with holes, and filled text.

Vertices in a given face must be coplanar. There is no implied connectivity between faces.

Edge data for a shell is listed in the order implied by the face list. For example, in the first face, vertex0 to vertex1 specifies the first edge, vertex1 to vertex2 specifies the second edge, and so on until the last vertex of the face, which connects to the first vertex, as shown below.

If the same edge is used in two different faces, properties may conflict. In such cases, you can set one of the edges to be invisible or make the properties match for each edge.

The order of face data, if present, follows the ordering of the face list for the shell.

The following is an example of a shell with color data attached to edges and faces and visibility data attached to edges. The shell is composed of two triangles in different planes that share a common edge. The common edge has silhouette visibility. This means that when the HIDE command is in effect and the AutoCAD variable DISPSILH equals 1 (display silhouettes is on), the common edge between the faces is drawn only if both faces in the viewport are on the same side of the common edge. In this case, one face is behind the other, so it is not drawn:

```
Adesk::Boolean
AsdkShellSamp::worldDraw(AcGiWorldDraw* pW)
{
    // Fill the faces with the current color.
    //
    pW->subEntityTraits().setFillType(kAcGiFillAlways);

    // Create vertices.
    //
    Adesk::UInt32 numVerts = 4;
    AcGePoint3d *pVerts = new AcGePoint3d[numVerts];
    pVerts[0] = AcGePoint3d(0.0, 0.0, 0.0);
    pVerts[1] = AcGePoint3d(0.0, 1.0, 0.0);
    pVerts[2] = AcGePoint3d(1.0, 1.0, 0.0);
    pVerts[3] = AcGePoint3d(1.0, 0.0, 2.0);
```

```
// Create two faces.
//
Adesk::UInt32 faceListSize = 8;
Adesk::Int32 *pFaceList
    = new Adesk::Int32[faceListSize];

// Assign vertices for face 1.
//
pFaceList[0] = 3;   // Three vertices in the face
pFaceList[1] = 0;   // pVerts[0]
pFaceList[2] = 1;   // pVerts[1]
pFaceList[3] = 2;   // pVerts[2]

// Assign vertices for face 2.
//
pFaceList[4] = 3;   // Three vertices in the face
pFaceList[5] = 0;   // pVerts[0]
pFaceList[6] = 2;   // pVerts[2]
pFaceList[7] = 3;   // pVerts[3]

// Apply colors to edges.
//
AcGiEdgeData edgeData;
int numEdges = 6;
short *pEdgeColorArray = new short[numEdges];
pEdgeColorArray[0] = kRed;
pEdgeColorArray[1] = kYellow;
pEdgeColorArray[2] = kGreen;
pEdgeColorArray[3] = kCyan;
pEdgeColorArray[4] = kBlue;
pEdgeColorArray[5] = kMagenta;
edgeData.setColors(pEdgeColorArray);

// Apply visibility to edges and make the common edge
// between two faces have silhouette visibility during
// the HIDE command with AutoCAD variable DISPSILH = 1.
//
Adesk::UInt8 *pEdgeVisArray
    = new Adesk::UInt8[numEdges];

edgeData.setVisibility(pEdgeVisArray);
pEdgeVisArray[0] = kAcGiVisible;
pEdgeVisArray[1] = kAcGiVisible;
pEdgeVisArray[2] = kAcGiSilhouette;
pEdgeVisArray[3] = kAcGiSilhouette;
pEdgeVisArray[4] = kAcGiVisible;
pEdgeVisArray[5] = kAcGiVisible;

// Apply colors to faces.
//
AcGiFaceData faceData;
int numFaces = 2;
short *pFaceColorArray = new short[numFaces];
```

```
    pFaceColorArray[0] = kBlue;
    pFaceColorArray[1] = kRed;
    faceData.setColors(pFaceColorArray);
    pW->geometry().shell(numVerts, pVerts, faceListSize,
        pFaceList, &edgeData, &faceData);
    delete [] pVerts;
    delete [] pFaceList;
    delete [] pEdgeColorArray;
    delete [] pFaceColorArray;

    return Adesk::kTrue;
}
```

An `AcGiVertexData` object contains a single flag that specifies how vertices in a shell are ordered. This flag is set and queried with the following functions:

```
virtual void
AcGiVertexData::setOrientationFlag(
    AcGiOrientationType oflag);
```

```
virtual AcGiOrientationType
AcGiVertexData::orientationFlag() const;
```

This flag is not used for meshes because the ordering of vertices specifying a mesh is fixed. Values for the flag are

- `kAcGiClockwise`
- `kAcGiCounterClockwise`
- `kAcGiNoOrientation`

The orientation of vertices in a shell's face list indicates the visible side of the face. For example, if the vertices are specified as clockwise and the vertices for a given face are listed in clockwise order, then that face is visible. In this case, faces with vertices in counterclockwise order are invisible.

Arc

The `circularArc()` function has two forms:

```
virtual Adesk::Boolean
AcGiWorldGeometry::circularArc(
    const AcGePoint3d& center,
    const double radius,
    const AcGeVector3d& normal,
    const AcGeVector3d& startVector,
    const double sweepAngle,
    const AcGiArcType arcType = kAcGiArcSimple) const = 0;
```

```
virtual Adesk::Boolean
AcGiWorldGeometry::circularArc(
    const AcGePoint3d& start,
    const AcGePoint3d& point,
    const AcGePoint3d& end,
    const AcGiArcType arcType = kAcGiArcSimple) const = 0;
```

The arc type variable, `AcGiArcType`, can have one of the following values:

- `kAcGiArcSimple` The arc itself, which is not fillable
- `kAcGiArcSector` The area bounded by the arc and its center of curvature
- `kAcGiArcChord` The area bounded by the arc and its end points

Simple Arc Arc Sector Arc Chord

Polyline

The `pline()` function allows a custom entity to draw graphics primitives using an `AcDbPolyline` as a template:

```
virtual Adesk::Boolean pline(
const AcDbPolyline& lwBuf,
    Adesk::UInt32 fromIndex = 0,
    Adesk::UInt32 numSegs = 0) const;
```

`AcDbPolylines` are multisegmented and support straight and curved segments with or without width. Using `pline()` provides the ability to generate contiguous straight and curved segments with width. None of the other AcGi primitive functions support width, so without using `pline()` it would be necessary to generate many parallel arc and line segments to simulate a filled arc or line segment with width. This is inefficient, and proper display is dependent upon the view (magnification).

Text

The example in this section shows use of the `AcGiTextStyle` class. It draws a rectangle around a piece of AcGi text that can be oriented and located anywhere in space.

The normal and direction vectors of the text must be perpendicular to each other. If you're unsure of the directions, consider the direction to be along the *X* axis and the normal along the *Z* axis in a right-handed coordinate system. Calculate the *Y* axis from these. Then the cross product of the *Y* axis to

Z axis will give you the normal plane's interpretation of the direction. Be sure that the direction is not aligned with the normal, or you will not have a direction with respect to the normal.

The `AcGiTextStyle::loadStyleRec()` function loads a font if it is not already loaded. (This function does not load an ACAD STYLE.) Its return values are as follows:

0x10	Another file (not FONTALT) opened in place of BigFont file name
0x08	Another file (not FONTALT) opened in place of file name
0x04	BigFont file name failed to be loaded
0x02	File name failed to be loaded
0x01	Files opened as called for

Text can be scaled in a number of ways. Use `AcGiTextStyle::setTextSize()` to scale the width and height of the text at the same time. Use `setXScale()` to scale the width of the text. Use `setTrackingPercent()` to specify how the characters of a particular font are placed next to each other. If you specify a value of 1.0, the spacing does not change; if you specify less than 1.0, the characters will squeeze together; and if it's more than 1.0, the characters will be farther apart. This example sets the tracking percent to a value of .80.

The `AcGiTextStyle::extents()` function returns the world coordinate size of the text's bounding box. If the `penups` argument is `kTrue`, then any undrawn pen moves made while the user was drawing the text will be included in the bounding box. The `raw` option tells the calculation to ignore escape code processing (so that "%%%" would not be interpreted as a single percent sign but as three percent signs).

The following example draws text and then draws a bounding box around a portion of the text.

```
Adesk::Boolean
AsdkTextStyleSamp::worldDraw(AcGiWorldDraw* pW)
{
    AcGePoint3d pos(4.0, 4.0, 0.0);
    AcGeVector3d norm(0.0, 0.0, 1.0);
    AcGeVector3d dir(-1.0, -0.2, 0.0);

    char *pStr = "This is a percent, '%%%'.";
    int len = strlen(pStr);
```

```
AcGiTextStyle style;
AcGeVector3d vec = norm;

vec = vec.crossProduct(dir);
dir = vec.crossProduct(norm);
style.setFileName("txt.shx");
style.setBigFontFileName("");
int status;

if (!((status = style.loadStyleRec()) & 1))
    pStr = "Font not found.";

pW->geometry().text(pos, norm, dir, pStr, len,
    Adesk::kFalse, style);
pos.y += 2.0;
style.setTrackingPercent(80.0);
style.setObliquingAngle(10.0);
AcGePoint2d ext = style.extents(pStr, Adesk::kFalse,
    strlen(pStr), Adesk::kFalse);
pW->geometry().text(pos, norm, dir, pStr, len,
    Adesk::kFalse, style);

// Draw a rectangle around the last text drawn.
// First, create a polyline the size of the
// bounding box. Then, transform it to the
// correct orientation, and then to the location of the
// text.

// Compute the matrix that orients the box.
//
AcGeMatrix3d textMat;
norm.normalize();
dir.normalize();

AcGeVector3d yAxis = norm;
yAxis = yAxis.crossProduct(dir);
yAxis.normalize();
textMat.setCoordSystem(AcGePoint3d(0.0, 0.0, 0.0), dir,
    yAxis, norm);

// Create the bounding box and enlarge it a little.
//
double offset = ext.y / 2.0;
AcGePoint3d verts[5];
verts[0] = verts[4] = AcGePoint3d(-offset, -offset, 0.0);
verts[1] = AcGePoint3d(ext.x + offset, -offset, 0.0);
verts[2] = AcGePoint3d(ext.x + offset, ext.y + offset, 0.0);
verts[3] = AcGePoint3d(-offset, ext.y + offset, 0.0);

// Orient and then translate each point in the
// bounding box.
//
```

```
    for (int i = 0; i < 5; i++) {
        verts[i].transformBy(textMat);
        verts[i].x += pos.x;
        verts[i].y += pos.y;
        verts[i].z += pos.z;
    }

    pW->geometry().polyline(5, verts);
    return Adesk::kTrue;
}
```

Associating an AcDbTextStyleTableRecord with an AcGiTextStyle

There are several global utility functions available to copy text style data from an `AcDbTextStyleTableRecord` to an `AcGiTextStyle`, and vice versa. The following functions use the object ID of the `AcDbTextStyleTableRecord` and a reference to the `AcGiTextStyle` to identify the two objects involved:

```
Acad::ErrorStatus
fromAcDbTextStyle(
    AcGiTextStyle& textStyle,
    const AcDbObjectId& AcDbStyleId);


Acad::ErrorStatus
toAcDbTextStyle(
    const AcDbObjectId AcDbStyleId,
    AcGiTextStyle& textStyle);
```

The following functions make use of the `AcGiTextStyle` and `AcDbTextStyleTableRecord` names:

```
Acad::ErrorStatus
fromAcDbTextStyle(
    AcGiTextStyle& textStyle,
    const char* AcDbStyleName);


Acad::ErrorStatus
toAcDbTextStyle(
    AcGiTextStyle& textStyle);


Acad::ErrorStatus
toAcDbTextStyle(
    AcGiTextStyle& textStyle,
    const char* AcDbStyleName);
```

When copying data to or from an `AcDbTextStyleTableRecord` that has been specified by name, the `AcGiTextStyle` object's name is set to match the name of the `AcDbTextStyleTableRecord`. If no record is found when copying to an `AcDbTextStyleTableRecord` specified by name, then one is created.

When copying from an AcGiTextStyle to an AcDbTextStyleTableRecord and the name of the AcGiTextStyle is used as the name of the AcDbTextStyleTableRecord, if the AcGiTextStyle does not have a name, a unique name is generated and used as the name for the AcGiTextStyle and AcDbTextStyleTableRecord objects.

The following functions are similar to the previous functions, except that they have an AcDbObjectId argument used for the objectId of the AcDbTextStyleTableRecord that the data has been copied into.

```
Acad::ErrorStatus
toAcDbTextStyle(
    AcGiTextStyle& textStyle,
    AcDbObjectId& AcDbStyleId);
```

```
Acad::ErrorStatus
toAcDbTextStyle(
    AcGiTextStyle& textStyle,
    const char* AcDbStyleName,
    AcDbObjectId& AcDbStyleId);
```

Using Drawables in Your Object

In addition to the predefined primitives, you can leverage abstract objects in your worldDraw() and viewportDraw() implementations. Any object that supports the AcGiDrawable protocol can be used as a sub-object. Since AcDbObject derives from AcGiDrawable, any database object can be used in this way, if it makes sense.

A drawable need not be graphical. For example, a drawable might only set the subentity traits to a particular combination and not implement worldDraw() or viewportDraw() at all.

This architecture allows the developer to distribute rendering code more efficiently. For example, the worldDraw() implementation of AcDbBlockReference simply pushes the insertion transform onto the stack, calls draw() passing in a pointer to its AcDbBlockTableRecord, and pops the matrix stack. The worldDraw() of AcDbBlockTableRecord iterates over the entities in the definition. The benefit of this technique is that the graphics for a given entity need only be generated once, and can be cached on the object and efficiently reused for subsequent references.

WARNING! Any drawables passed into `draw()` must have a lifetime equal to or exceeding that of the outer object. This is required because the graphics of a drawable might be cached in an `AcGsNode` attached to the drawable. During display the graphics system might go back to get this cache, and if the object has been destroyed, a runtime error will occur.

Tessellation

Curves and curved surfaces need to be tessellated—broken up into lines and polygons—in order to be displayed. The degree of tessellation determines how accurate the displayed curve will be (how close it will approximate the mathematical "true" curve) and how much performance overhead is required to generate the graphics for a curve. A very small circle may require only a single pixel to display it. A large circle may require hundreds of small line segments to be calculated and displayed to create a smooth appearance.

The `deviation()` functions provided by the `AcGiWorldDraw` and `AcGiViewportDraw` classes return the deviation, which is the allowable maximum difference in world space between a true mathematical surface and the tessellated surface, as shown in the following figure:

d = maximum deviation

Access to this value allows custom entities to tune their tessellation to the VIEWRES command's zoom percent option, which is set by the user. The result is that custom entities are tessellated to relatively the same smoothness as built-in entities.

The `deviation()` function returns the suggested maximum deviation in world space, given the type of deviation to calculate and a point in world space for perspective scaling if required. The signature for the `deviation()` function is

```
virtual double
AcGiWorldDraw::deviation(
    AcGiDeviationType devType,
    const AcGePoint3d&) const = 0;
```

The deviation types are

- `kAcGiMaxDevForCircle` (for circles and arcs)
- `kAcGiMaxDevForCurve`
- `kAcGiMaxDevForBoundary`
- `kAcGiMaxDevForIsoline`
- `kAcGiMaxDevForFacet` (for surfaces; formula for calculating this deviation uses the value of the FACETRES system variable)

Isolines

An isoline is used to give a visual clue to the shape of an object. The `AcGiWorldDraw::isolines()` function allows an entity to display the same number of isolines per surface as specified by the user. This value is an integer between 0 and 2047. The default number of isolines is 4. The `AcGiViewportDraw` class provides an analogous function:

```
virtual Adesk::UInt32
AcGiWorldDraw::numberOfIsolines() const;
```

Transformations

The graphics pipeline can apply three possible transformations to an entity:

- The entity's block transformations
- The viewport's view transformation
- The perspective transformation (if perspective is enabled from DVIEW)

Each transformation produces a new type of coordinates, as shown in the following figure. If not in perspective mode, eye and display coordinates are identical.

For the REGEN, HIDE, and SHADE commands, the entity's world coordinates are sent through the graphics pipeline shown in the figure above. The view transformation specifies a particular view of the world coordinates, analogous to viewing a scene with a camera. The camera has a location in world space and a particular orientation toward the world coordinate "scene." When the view transformation is complete, world coordinates are transformed to eye coordinates, looking down the *Z* axis of the camera.

If perspective is enabled, the eye coordinates are transformed to display coordinates. This transformation involves division according to how far away something is from the camera, so that objects farther away from the camera appear smaller than objects closer to the camera.

The following sections discuss these coordinate systems in greater detail.

Model Coordinate System

Model coordinates are those coordinates that have not yet been transformed by any block inserts or views. One piece of model coordinate geometry can be contained in an unlimited number of block inserts. Most of the AcGi geometry methods use model coordinates. However, there are a few geometry methods that accept other coordinate systems.

When converting an entity's model coordinate geometry to world coordinates, the current net block transform is used. For example, if a piece of model coordinate geometry is in more than one block insert, then the net effect of being in the inserts is the net block transform.

World Coordinate System

The World Coordinate System (WCS) is the "reference" coordinate system. The WCS is a fixed coordinate system; all other coordinate systems are defined relative to the WCS. This is the coordinate system in which all of the model coordinate geometry has been placed according to its block insert. If any of the model coordinate geometry was not included in a block insert, then the model and world coordinates are identical.

Eye Coordinate System

Every viewport has a vantage point, tilt, aim, and a field of view on the WCS. This is summed up in one transform, the viewport's view transform. This transform converts world coordinate geometry into view coordinate geometry where the viewport's view is down its *Z*-axis looking toward more negative values of *Z*. Eye coordinates do not contain the perspective transform.

Display Coordinate System

The Display Coordinate System (DCS) is the coordinate system into which objects are transformed before they are displayed. If not in perspective mode, the DCS is equivalent to the eye coordinate system. If perspective mode is active, the perspective transform must be performed on the eye coordinates to get display coordinates. The perspective transform gives the effect of geometry getting smaller the farther away it is from the camera.

Transformation Examples

The `AcGiViewport` class provides functions that give you access to the graphics pipeline, allowing you to apply each transformation explicitly and perform the mathematics yourself. If you are manipulating entities in the graphics pipeline yourself, you use different forms of the AcGi polygon and polyline depending on where you are in the graphics pipeline.

The `AcGiViewportGeometry` class provides three forms for polygons and polylines, in model, eye, and display coordinates. Normally, you would use the `polyline()` and `polygon()` functions, which require model coordinates. Use `polylineEye()` and `polygonEye()` if you are going to work with eye coordinates, as shown in Examples 1 and 2. Use `polygonDc()` and `polylineDc()` if you are working with display coordinates.

The following sections contain four examples. The first example draws the same entity using model, eye, and display coordinates. Its main purpose is to demonstrate how to apply each transformation in the graphics pipeline. The second example illustrates working with eye coordinates to determine the front and back faces of a pyramid. The third example illustrates working with display coordinates to draw an entity in a size relative to the size of the current window. The fourth example shows how to determine the polyline with the fewest segments that is visually indistinguishable from one with more segments.

Example 1: Coordinate Systems

This example takes a line segment defined in model coordinates and creates its equivalents in eye and display coordinates. When displayed, all lines will overlap.

```
void
AsdkCoordSamp::viewportDraw(AcGiViewportDraw* pV)
{
    pV->subEntityTraits().setFillType(kAcGiFillAlways);
    const int count = 3;
    AcGePoint3d verts[count];

    verts[0] = AcGePoint3d(0.0, 0.0, 0.0);
    verts[1] = AcGePoint3d(1.0, 0.0, 0.0);
    verts[2] = AcGePoint3d(1.0, 1.0, 0.0);

    // Draw model space line segment.
    //
    pV->subEntityTraits().setColor(kBlue);
    pV->geometry().polygon(count, verts);

    // Compute the line's representation in eye space.
    //
    AcGeMatrix3d mat;
    pV->viewport().getModelToEyeTransform(mat);

    for (int i = 0; i < count; i++) {
        verts[i].x += 0.01;
        verts[i].y += 0.01;
        verts[i].z += 0.01;
        verts[i].transformBy(mat);
    }
```

```
    // Display the eye coordinate equivalent of the
    // model space polygon.
    //
    pV->subEntityTraits().setColor(kGreen);
    pV->geometry().polygonEye(count, verts);

    // Convert from eye to display coordinates.
    //
    for (i = 0; i < count; i++) {
        verts[i].x += 0.01;
        verts[i].y += 0.01;
        verts[i].z += 0.01;
    }

    // Draw the display space equivalent of the
    // model space polygon.
    //
    pV->subEntityTraits().setColor(kRed);
    pV->geometry().polygonDc(count, verts);
}
```

Example 2: Determining Hidden Lines for an Object for Standard Display

This example displays a pyramid, showing the front edges in yellow and the back edges in blue to give you an idea of the visible and hidden edges of the pyramid. The example shows applying the model-to-eye transformation and then the perspective transformation. It uses eye coordinates to draw the entity and shows use of `isPerspective()`, `doPerspective()`, `getFrontandBackClipValues()`, `polylineDc()`, `polylineEye()`, and `polyline()`.

To convert the eye-coordinate line segment to display space

1. If a view has clipping planes in force, clip the eye-coordinate line segment to them.
2. If perspective is on, then perform the conversion from eye coordinates to perspective.

If you're using the `polygonEye()`, `polygonDc()`, `polylineEye()`, or `polylineDc()` functions of `AcGiViewportGeometry`, you should call `AcGiWorldGeometry::setExtents()` to establish the bounding box for the entity. This will let AutoCAD know how much space the entity requires and is used in ZOOM Extents. The `setExtents()` function is usually called when the entity is in world coordinates to determine the smallest box that will fit around the entity in world coordinates.

```
AsdkViewGeomSamp::AsdkViewGeomSamp() : mNumVerts(4)
{
    mVerts[0] = AcGePoint3d(0.0, 0.0, 0.0);
    mVerts[1] = AcGePoint3d(1.0, 0.0, 0.0);
    mVerts[2] = AcGePoint3d(0.0, 1.0, 0.0);
    mVerts[3] = AcGePoint3d(0.0, 0.0, 1.0);
}

Acad::ErrorStatus
AsdkViewGeomSamp::transformBy(const AcGeMatrix3d &xfm)
{
    assertWriteEnabled();
    for (Adesk::UInt32 i = 0; i < mNumVerts; i++) {
        mVerts[i].transformBy(xfm);
    }
    return Acad::eOk;
}

Adesk::Boolean
AsdkViewGeomSamp::worldDraw(AcGiWorldDraw* pW)
{
    // Draw a pyramid.
    //
    // If this is the regular AutoCAD DISPLAY mode...
    //
    if (pW->regenType() == kAcGiStandardDisplay) {
        //  From each viewport's vantage point, figure out
        //  which sides of the pyramid are visible,
        //  then make the visible ones yellow and the hidden
        //  ones blue.
        //
        //  Set the extents of the pyramid here because
        //  AcGiViewportGeometry's polylineEye() doesn't
        //  set extents.
        //
        for (Adesk::UInt32 i = 0; i < mNumVerts; i++) {
            AcGePoint3d pt[2];
            pt[0] = mVerts[i];
            pt[1] = mVerts[(i + 1) % mNumVerts];
            pW->geometry().setExtents(pt);
        }
        return Adesk::kFalse;  // Call viewport draws.
    }
```

```
    // Otherwise, give HIDE, SHADE, RENDER, or proxy graphics
    // a pyramid with filled faces.
    //
    const Adesk::UInt32 faceListSize = 16;
    static Adesk::Int32 faceList[faceListSize] = {
        3, 0, 1, 2,
        3, 0, 2, 3,
        3, 0, 3, 1,
        3, 1, 2, 3
    };

    pW->geometry().shell(mNumVerts, mVerts, faceListSize,
        faceList);
    return Adesk::kTrue;  // Do not call viewportDraw.
}

void
AsdkViewGeomSamp::viewportDraw(AcGiViewportDraw* pV)
{
    // For this viewport, draw a pyramid with yellow
    // visible lines and blue hidden lines.
    //
    // Get this viewport's net transform.  This transform
    // includes this entity's block transforms and this
    // viewport's view transform; it does not include the
    // perspective transform if we're in perspective
    // mode; that currently has to be applied separately
    // when in perspective mode.
    //
    AcGeMatrix3d modelToEyeMat;
    pV->viewport().getModelToEyeTransform(modelToEyeMat);

    // Get the pyramid's vertices.
    //
    AcGePoint3d A = mVerts[0];
    AcGePoint3d B = mVerts[1];
    AcGePoint3d C = mVerts[2];
    AcGePoint3d D = mVerts[3];

    // Convert them to the viewport's eye coordinates.
    //
    A.transformBy(modelToEyeMat);
    B.transformBy(modelToEyeMat);
    C.transformBy(modelToEyeMat);
    D.transformBy(modelToEyeMat);

    // Save the eye coordinates.
    //
    AcGePoint3d AEye = A;
    AcGePoint3d BEye = B;
    AcGePoint3d CEye = C;
    AcGePoint3d DEye = D;
```

```
// Perform the perspective transform if necessary.
//
if (pV->viewport().isPerspective()) {
    pV->viewport().doPerspective(A);
    pV->viewport().doPerspective(B);
    pV->viewport().doPerspective(C);
    pV->viewport().doPerspective(D);
}

// From that view, figure out which faces are
// facing the viewport and which are not.
//
int which_faces;
which_faces  = ((C - A).crossProduct(B - A)).z
    > 0.0 ? 1 : 0;
which_faces |= ((D - A).crossProduct(C - A)).z
    > 0.0 ? 2 : 0;
which_faces |= ((B - A).crossProduct(D - A)).z
    > 0.0 ? 4 : 0;
which_faces |= ((B - D).crossProduct(C - D)).z
    > 0.0 ? 8 : 0;

// Those edges that meet between two faces that are
// facing away from the viewport will be hidden edges,
// so draw them blue; otherwise, they are visible
// edges.  (This example is incomplete, as the test is
// indeterminate when the face is edge-on to the
// screen -- neither facing away nor toward the screen.)
// Draw the six edges connecting the vertices using eye
// coordinate geometry that can be clipped back and front.
//
AcGePoint3d verts[2];
Adesk::UInt16 color;

// AB
color = which_faces & 0x5 ? kYellow : kBlue;
pV->subEntityTraits().setColor(color);
verts[0] = AEye;
verts[1] = BEye;
pV->geometry().polylineEye(2, verts);

// AC
color = which_faces & 0x3 ? kYellow : kBlue;
pV->subEntityTraits().setColor(color);
verts[0] = AEye;
verts[1] = CEye;
pV->geometry().polylineEye(2, verts);

// AD
color = which_faces & 0x6 ? kYellow : kBlue;
pV->subEntityTraits().setColor(color);
verts[0] = AEye;
verts[1] = DEye;
pV->geometry().polylineEye(2, verts);
```

```
    // CD
    color = which_faces & 0xa ? kYellow : kBlue;
    pV->subEntityTraits().setColor(color);
    verts[0] = CEye;
    verts[1] = DEye;
    pV->geometry().polylineEye(2, verts);

    // DB
    color = which_faces & 0xc ? kYellow : kBlue;
    pV->subEntityTraits().setColor(color);
    verts[0] = DEye;
    verts[1] = BEye;
    pV->geometry().polylineEye(2, verts);

    // BC
    color = which_faces & 0x9 ? kYellow : kBlue;
    pV->subEntityTraits().setColor(color);
    verts[0] = BEye;
    verts[1] = CEye;
    pV->geometry().polylineEye(2, verts);
}
```

Example 3: Obtaining the Window Coordinates

This example shows the use of `AcGiViewportDraw::polylineDc()` and `AcGiViewport::getViewportDcCorners()` to obtain the display coordinates of the viewport. This function is convenient when you are drawing graphics that depend on the physical layout of the viewport, such as icons and markers that vary with the size or borders of the viewport. The graphics drawn will change with panning and zooming.

The example draws a box in the upper-right corner of the viewport. The box's width and height are always one-tenth of the viewport's shortest dimension, and the box is centered one-tenth of the viewport's shortest dimension down and to the left of the upper-right-hand corner of the viewport:

```
void
AsdkIconSamp::viewportDraw(AcGiViewportDraw* pV)
{
    // Get the current viewport's display coordinates.
    //
    AcGePoint2d lower_left, upper_right;
    pV->viewport().getViewportDcCorners(lower_left,
        upper_right);

    double xsize = upper_right.x - lower_left.x;
    double ysize = upper_right.y - lower_left.y;
    xsize /= 10.0;
    ysize /= 10.0;

    double xcenter = upper_right.x - xsize;
    double ycenter = upper_right.y - ysize;
    double half_xsize = xsize / 2.0;
    double half_ysize = ysize / 2.0;
```

```
    // Create a unit square.
    //
    const int num_verts = 5;
    AcGePoint3d verts[num_verts];

    for (int i = 0; i < num_verts; i++) {
        verts[i].x = xcenter;
        verts[i].y = ycenter;
        verts[i].z = 0.0;
    }

    verts[0].x -= half_xsize;
    verts[0].y += half_ysize;
    verts[1].x += half_xsize;
    verts[1].y += half_ysize;
    verts[2].x += half_xsize;
    verts[2].y -= half_ysize;
    verts[3].x -= half_xsize;
    verts[3].y -= half_ysize;
    verts[4] = verts[0];

    pV->subEntityTraits().setColor(kRed);

    pV->geometry().polylineDc(num_verts, verts);
}
```

Example 4: Calculating the Circle to Draw

The following example draws a unit circle centered at the origin. The exact circle drawn depends on the viewport's view of the circle. The objective is to draw a circle with a polyline that has the minimum number of discernible segments. With the VPORTS command, you can create four viewports and then click on one and zoom in on the circle, then click on another and back up from it. When you do a REGENALL, each viewport calculates its own minimally segmented polyline representation of the circle.

This is how the example calculates the necessary number of line segments in the polyline. First, given a circle of a given radius that is centered at the origin and located in the *XY* plane, and given a vertical line that intersects the *X* axis at *radius – 0.5 pixels*, determine the angle between the *X* axis and a line segment that extends from the origin to the point where the vertical line intersects the circle. Two pi divided by this angle provides the minimum number of segments needed by a polyline to look like a circle. The user will not be able to differentiate the individual line segments that make up the circle because the visual differences are less than a pixel.

```
Adesk::Boolean
AsdkTesselateSamp::worldDraw(AcGiWorldDraw *pW)
{
    // Draw a red 1x1 drawing-unit square centered at the
    // world coordinate origin and parallel to the XY-plane.
    //
    const Adesk::UInt32 num_pts = 5;
    AcGePoint3d verts[num_pts];

    verts[0] = verts[4] = AcGePoint3d(-0.5, -0.5, 0.0);
    verts[1] = AcGePoint3d( 0.5, -0.5, 0.0);
    verts[2] = AcGePoint3d( 0.5,  0.5, 0.0);
    verts[3] = AcGePoint3d(-0.5,  0.5, 0.0);
    pW->subEntityTraits().setColor(kRed);
    pW->geometry().polyline(num_pts, verts);

    // If regenType is kAcGiSaveWorldDrawForProxy, return
    // Adesk::kTrue, otherwise return Adesk::kFalse to trigger
    // calls to viewportDraw().
    //
    return (pW->regenType() == kAcGiSaveWorldDrawForProxy);
}

void
AsdkTesselateSamp::viewportDraw(AcGiViewportDraw *pV)
{
    static double two_pi = atan(1.0) * 8.0;
```

```
// Get the number of pixels on the X- and Y-edges of
// a unit square centered at (0.0, 0.0, 0.0), in
// world coordinates.
//
AcGePoint3d center(0.0, 0.0, 0.0);
AcGePoint2d area;
pV->viewport().getNumPixelsInUnitSquare(center, area);

// If the area values are negative, then we are in
// perspective mode and the center is too close or
// in back of the viewport.
//
if (area.x > 0.0) {
    // Print out the number of pixels along the
    // Y-axis of the unit square used in
    // getNumPixelsInUnitSquare.
    //
    AcGeVector3d norm(0.0, 0.0, 1.0);
    AcGeVector3d dir(1.0, 0.0, 0.0);
    char buf[100];
    sprintf(buf, "%7.3lf", area.y);

    pV->geometry().text(center, norm, dir, 1.0, 1.0,
        0.0, buf);

    // Draw a circle that depends on how big the circle
    // is in the viewport.  This requires
    // figuring out the fewest number of segments needed
    // by a polyline so that it doesn't look segmented.
    //
    // The worldDraw() and viewportDraw() of
    // an entity in a viewport are only called during a
    // regen and not necessarily during a ZOOM or PAN.
    // The reason is that a regen produces something
    // akin to a very high resolution image internally,
    // which AutoCAD can zoom in or pan around. That is,
    // until you get too close to this image or any of
    // its edges, at which point a regen is internally
    // invoked for that viewport and a new internal
    // image is created (ready to be mildly zoomed and
    // panned).
    //
    double radius = 0.5;
    double half_pixel_hgt = 2.0 / area.x; // In WCS

    int num_segs = 8;
    double angle = two_pi / num_segs;

    if (half_pixel_hgt > radius / 2) {
        // The circle is approximately the same or less
        // than the size of a pixel. So, generate a very
        // small octagon.
        //
        num_segs = 8;
    } else {
```

```
            // Given a circle centered at the origin of a
            // given radius in the XY-plane, and given a
            // vertical line that intersects the X-axis at
            // 'radius - half a pixel', what is the angle
            // from the X-axis of a line segment from the
            // origin to the point where the vertical line
            // and the circle intersect?  Two pi divided by
            // this angle gives you a minimum number of
            // segments needed by a polyline to look like
            // a circle and not be able to differentiate
            // the individual segments because the visual
            // differences are less than the size of a
            // pixel.  (This is not the only way to figure
            // this out but it's sufficient.)
            //
            angle = acos((radius - 1.0 / (area.x / 2.0))
                / radius);
            double d_num_segs = two_pi / angle;

            // Limit the number of segments from 8 to
            // 128 and use whole numbers for
            // this count.
            //
            if (d_num_segs < 8.0) {
                num_segs = 8;
            } else if (d_num_segs > 128.0) {
                num_segs = 128;
            } else {
                num_segs = (int)d_num_segs;
            }
        }

        // Calculate the vertices of the polyline from the
        // start, around the circle, and back to the start
        // to close the polyline.
        //
        angle = 0.0;
        double angle_inc = two_pi / (double)num_segs;
        AcGePoint3d* verts = new AcGePoint3d[num_segs + 1];

        for (int i = 0; i <= num_segs; i++,
            angle += angle_inc)
        {
            verts[i].x = center.x + radius * cos(angle);
            verts[i].y = center.y + radius * sin(angle);
            verts[i].z = center.z;
        }
        pV->geometry().polyline(num_segs + 1, verts);
        delete [] verts;
    }
}
```

Using Clip Boundaries in AcGi

ObjectARX allows you to define a clipping boundary for geometry contained within xrefs and blocks. The clip-boundary portion of the AcGi API allows compound objects (blocks and objects that behave like blocks) to express these clip boundaries to the AcGi implementation.

During `worldDraw()` or `viewportDraw()` any drawable may specify a polygonal clip boundary with which to clip its graphics. The following sections describe how to use this API feature.

Background

Clip boundaries are closed, non-self-intersecting, concave 2D polygons. Optional front and back Z clipping values can be assigned. The clip boundary is expressed in an arbitrary coordinate system relative to the objects being clipped.

In AutoCAD, when the user defines a clipping boundary for a block, the view direction and twist of the current view are used to define the coordinate system for the clip boundary. This might be the same as the coordinate system of the block reference being clipped. This is reflected in the API by the provision of a transformation to the clipping space from the block reference system:

Clip boundaries can be nested. A compound object can define a clipping boundary, and the objects that it contains can also define boundaries for their internal geometry. In this case, the nested geometry is first clipped against its parent's boundary and any resultant fragments are then clipped against the clip boundary of the outer block.

Clip Boundary Example

In the following example, the clip boundary is pushed onto the clip boundary stack before anything is drawn and popped off again once the drawing for this object is complete:

```
Adesk::Boolean
MyObject::worldDraw(AcGiWorldDraw* pDraw)
{
    AcGiWorldGeometry * pGeom = &pDraw->geometry();

    pGeom->pushModelTransform(myTransform());

    AcGiClipBoundary cb;
    cb.m_bDrawBoundary= true;
    cb.m_vNormal = AcGeVector3d::kZAxis;
    cb.m_ptPoint = AcGePoint3d::kOrigin;

    // Two points treated as a rectangle, three creates a triangle
    cb.m_aptPoints.append(AcGePoint2d(0,0));
    cb.m_aptPoints.append(AcGePoint2d(5,5));

    // We are clipping in our own space
    cb.m_xToClipSpace.setToIdentity();
    cb.m_xInverseBlockRefXForm = myTransform().inverse();

    // No Z clipping
    cb.m_bClippingBack = cb.m_bClippingFront = false;
    cb.m_dFrontClipZ = cb.m_dBackClipZ = 0.;

   Adesk::Boolean bPopClipBoundary = pGeom->pushClipBoundary(&cb);

    // Draw something
    pGeom->circle(...);

    pGeom->popModelTransform();
    if(bPopClipBoundary){ pGeom->popClipBoundary(); }
    return true; // world-only
}
```

Since this clipping is a complex operation, some AcGi implementations might not support it fully. In this case, the AcGi implementation may return false from `pushClipBoundary()`, and you should not call `popClipBoundary()`.

Using the Geometry Library

In This Chapter

27

- Overview of the AcGe Library
- Using Basic Geometry Types
- Using the Line and Plane Classes
- Parametric Geometry
- Special Evaluation Classes
- Persistent AcGe Entities

This chapter discusses the main uses of the AcGe library, which provides a number of classes for representing 2D and 3D geometry. This library is intended for use by any Autodesk application and is frequently used by the AcDb and AcGi libraries in ObjectARX.

Overview of the AcGe Library

The AcGe library includes a broad set of classes for representing commonly used geometry, such as points, lines, curves, and surfaces. It provides a common representation for geometry that can be used by any Autodesk application. The library is purely mathematical; though its classes do not deal directly with the database or with graphics, many of its classes are used by the AcDb and AcGi libraries.

The class hierarchy for the AcGe library is shown as follows:

```
AcGeEntity2d
├── AcGeBoundBlock2d
├── AcGeClipBoundary2d
├── AcGeCurve2d
│   ├── AcGeCircArc2d
│   ├── AcGeCompositeCurve2d
│   ├── AcGeEllipArc2d
│   ├── AcGeExternalCurve2d
│   ├── AcGeLinearEnt2d
│   │   ├── AcGeLine2d
│   │   ├── AcGeLineSeg2d
│   │   └── AcGeRay2d
│   ├── AcGeOffsetCurve2d
│   └── AcGeSplineEnt2d
│       ├── AcGeCubicSplineCurve2d
│       ├── AcGeNurbCurve2d
│       └── AcGePolyline2d
├── AcGeCurveCurveInt2d
└── AcGePointEnt2d
    ├── AcGePointOnCurve2d
    └── AcGePosition2d

AcGeCurveBoundary
AcGe
AcGeContext
AcGeDwgIO
AcGeDxfIO
AcGeFileIO
AcGeFiler
AcGeInterval
AcGeKnotVector
AcGeLibVersion
AcGeMatrix2d
AcGeMatrix3d
AcGePoint2d
└── AcAxPoint2d
AcGePoint3d
└── AcAxPoint3d
AcGeScale2d
AcGeScale3d
AcGeTol
AcGeVector2d
AcGeVector3d
```

```
AcGeEntity3d
├── AcGeBoundBlock3d
├── AcGeCurve3d
│   ├── AcGeCircArc3de
│   ├── AcGeCompositeCurve3d
│   ├── AcGeEllipArc3e
│   ├── AcGeExternalCurve3d
│   ├── AcGeLinearEnt3d
│   │   ├── AcGeLine3d
│   │   ├── AcGeLineSeg3d
│   │   ├── AcGeRay3d
│   │   └── AcGeMatrix3d
│   ├── AcGeOffsetCurve3d
│   └── AcGeSplineEnt3d
│       ├── AcGeCubicSplineCurve3d
│       ├── AcGeNurbCurve3d
│       └── AcGePolyline3d
│           └── AcGeAugPolyline3d
├── AcGeCurveCurveInt3d
├── AcGeCurveSurfInt
├── AcGePointEnt3d
│   ├── AcGePointOnCurve3d
│   ├── AcGePointOnSurface
│   └── AcGePosition3d
├── AcGeSurfSurfInt
└── AcGeSurface
    ├── AcGeCone
    ├── AcGeCylinder
    ├── AcGeExternalBoundedSurface
    ├── AcGeExternalSurface
    ├── AcGeNurbSurface
    ├── AcGeOffsetSurface
    ├── AcGePlanarEnt
    │   ├── AcGeBoundedPlanet
    │   └── AcGePlane
    ├── AcGeSphere
    └── AcGeTorus
```

The AcGe library provides both simple and complex geometry classes. Simple linear algebra classes include the point, vector, matrix, 2D and 3D linear entity classes, and planar entity classes. Complex classes include curve classes, such as spline entity, and surface classes, such as NURBS surfaces.

The class hierarchy offers separate classes for 2D and 3D geometry. This simplifies programming by clearly distinguishing 2D parametric-space geometry from 3D modeling-space geometry. Because of this distinction, you cannot inadvertently mix 2D and 3D entities in the same operation.

The library includes a number of basic types, such as `AcGePoint3d`, `AcGeVector3d`, and `AcGeMatrix3d`, that have public data members for fast and efficient access. These simple classes are commonly used by other libraries as well as by the AcGe classes derived from `AcGeEntity2d` and `AcGeEntity3d`.

Runtime type checking is provided for all classes derived from `AcGeEntity2d` and `AcGeEntity3d`. Each class provides a `type()` function that returns the object's class and an `isKindOf()` function that returns whether the object is of a particular class (or a class derived from that class).

Two entities are considered equal if they are of the same type and represent the same point set. Curves and surfaces are considered equal only if their parameterization is the same.

Global Data and Functions

The following table lists the global identifiers defined by some of the header files.

Global identifiers and header files

Header File	Global Functions Defined
gepnt2d.h	AcGePoint2d::kOrigin
gemat2d.h	AcGeMatrix2d::kIdentity
gevec2d.h	AcGeVector2d::kIdentity AcGeVector2d::kXAxis AcGeVector2d::kYAxis
geline2d.h	AcGeLine2d::kXAxis AcGeLine2d::kYAxis
gepnt3d.h	AcGePoint3d::kOrigin
gemat3d.h	AcGeMatrix3d::kIdentity

Global identifiers and header files (*continued*)	
Header File	**Global Functions Defined**
gevec3d.h	AcGeVector3d::kIdentity AcGeVector3d::kXAxis AcGeVector3d::kYAxis AcGeVector3d::kZAxis
geline3d.h	AcGeLine3d::kXAxis AcGeLine3d::kYAxis AcGeLine3d::kZAxis
geplane.h	AcGePlane::kXYPlane AcGePlane::kYZPlane AcGePlane::kXZPlane
gegbl.h	AcGeContext::gOrthoVector()

`AcGeContext::gOrthoVector` is a pointer to a function that, given a vector, computes an arbitrary vector that is perpendicular to it. You can substitute your own function for the given function.

Tolerances

Many methods accept a tolerance value as one of their parameters. This value is of the `AcGeTol` class and always has a default value, as defined in `AcGeContext::gTol`. Functions such as `isClosed()` and `isPlanar()` calculate whether the start points and endpoints are within the defined tolerance before returning a Boolean value. You can change the tolerance for one particular function call, or you can change the global tolerance value.

The `AcGeTol` class provides two functions for setting the tolerance for points and vectors:

```
void
setEqualPoint(double);
```

```
void
setEqualVector(double);
```

The `AcGeTol` class also provides two functions for obtaining the tolerance for points and vectors:

```
double  equalPoint() const;
double  equalVector() const;
```

The `equalPoint` and `equalVector` tolerance values are used as follows:

- Two points, `p1` and `p2`, are equal if
  `(p1 - p2).length() <= equalPoint`
- Two vectors, `v1` and `v2`, are equal if
  `(v1 - v2).length() <= equalVector`
- Two vectors, `v1` and `v2`, are parallel if
  ```
  (v1/v1.length() - v2/v2.length()).length() < equalVector
   OR (v1/v1.length() + v2/v2.length()).length() < equalVector
  ```
- Two vectors, `v1` and `v2`, are perpendicular if
  ```
  abs((v1.dotProduct(v2))/(v1.length()*v2.length()))
      <= equalVector
  ```
- Two lines or rays are parallel (perpendicular) if their directional vectors are parallel (perpendicular)
- Two lines are equal if the points at parameter 0 are equal and their directions are equal

---

NOTE These rules mean that two lines are close to each other as point sets in the part of the modeling space of diameter *diam* only if the tolerance *equalVector* is set tighter than *equalPoint/diam*.

Using Basic Geometry Types

The following examples show some of the most commonly used functions and operators in the point, vector, and matrix classes. These examples use the 3D classes, but most of them also apply to the 2D classes as well.

The default constructor for points and vectors initializes all coordinates to 0. Points and vectors may also be constructed by specifying their coordinates as follows:

```
AcGePoint3d    p1(2.0,5.0,-7.5), p2, p3(1.0,2.0,3.0);
AcGeVector3d   v1(3.0,4.0,5.0), v2(0.0,1.0,-1.0), v3;
```

The point and vector classes provide `+`, `+=`, `-`, and `-=` operators. These operators allow points and vectors to be used in much the same way as built-in types, such as doubles and integers. The following are examples of adding and subtracting points and vectors:

```
p2 = p1 + v1;       // Set p2 to sum of p1 and v1.
p1 += v1;           // Add v1 to p1.
p3 -= v1;           // Subtract v1 from p3.
v3 = v1 + v2;       // Set v3 to sum of v1 and v2.
```

```
v1 += v2;           // Add v2 to v1.
v3 = v1 - v2;       // Set v3 to difference of v1 and v2.
```

There is no + operator for adding two points; however, a point can be converted to a vector, which can then be added to another point:

```
p1 += p2.asVector();
```

The following are examples of how to obtain the negative of a vector:

```
v2 = -v1;           // Set v2 to negative of v1.
v1.negate();        // This is equivalent to v1 = -v1.
```

The following are examples of different ways to scale a vector:

```
v1 *= 2.0;          // Doubles the length of v1.
v3 = v1 / 2.0;      // Set v3 to half the length of v1.
v1.normalize();     // Make v1 a unit vector.
```

The point and vector classes contain a number of query functions for computing distances and lengths:

```
double len = v2.length();  // Length of v2.
len = p1.distanceTo(p2);   // Distance from p1 to p2.
```

The following function is very useful for computing the angle between two 3D vectors. The following returns the angle between `v1` and `v2` where the angle is taken to be counterclockwise about `v3` (`v3` is assumed to be perpendicular to `v1` and `v2`):

```
angle = v1.angleTo(v2,v3);
```

The following functions return a Boolean value (TRUE or FALSE) and may be used inside `if` statements:

```
if (v1.isZeroLength())
if (v1.isParallelTo(v2))
if (v1.isPerpendicularTo(v2))
```

The vector class contains functions for the usual vector operations:

```
len = v1.dotProduct(v2);
v3 = v1.crossProduct(v2);
```

The default constructor for a matrix initializes the matrix to the identity matrix:

```
AcGeMatrix3d   mat1, mat2, mat3;
```

The following rotates `p3` 90 degrees about the line defined by `p1` and `v1`:

```
mat1.setToRotation ( kPi/2.0, v1, p1 );
p3 = mat1 * p2;
```

A matrix can be inverted if it is not singular:

```
if (!mat2.isSingular())
    mat2.invert();
```

The * operator is defined for concatenating matrices:

```
mat3 = mat1 * mat2;
```

The following tests whether a matrix contains equal scaling in all three coordinates (that is, it does not change the shape of any entity to which it is applied):

```
if (mat.isUniScaledOrtho())
```

Using the Line and Plane Classes

The following examples show some of the most commonly used functions in the line and plane classes. These examples show how to use the line and plane classes for basic linear algebra operations. Although the examples use the 3D classes, most of the functions that do not involve the plane class are also present in the 2D classes. These examples also use the infinite line and plane classes, but they are equally valid for line segments, rays, and bounded planes.

The default line constructor constructs a line along the *X* axis. The default plane constructor constructs the *XY* plane:

```
AcGePoint3d    p1(2.0,5.0,-7.5), p2;
AcGeLine3d     line1(p1,v1), line2;
AcGePlane      plane1(p1,v1), plane2;
```

The above constructor for `line1` constructs a line through `p1` in the direction of `v1`. The constructor for `plane1` constructs a plane through `p1` and normal to `v1`. Thus, `line1` is perpendicular to `plane1`.

The following functions return the line or plane definition:

```
p1 = line1.pointOnLine();   // Arbitrary point on line.
v1 = line1.direction();     // Direction vector of line.
p1 = plane1.pointOnPlane(); // Arbitrary point on plane.
v1 = plane1.normal();       // Normal vector of plane.
```

The `direction()` and `normal()` functions always return unit vectors.

The following functions return the closest point on the line or plane to the point `p1`:

```
p2 = line1.closestPointTo(p1);
p2 = plane1.closestPointTo(p1);
```

The following functions return the distance between a point and line or plane (these distances will be the same as the distances between `p1` and `p2` above):

```
double len = line1.distanceTo(p1);
len = plane1.distanceTo(p1);
```

The following functions return a Boolean value (TRUE or FALSE) and may be used inside an `if` statement. The first two test if the point `p1` lies on `line1` or `plane1`, and the third tests if `line1` lies on `plane1`:

```
if (line1.isOn(p1))
if (plane1.isOn(p1))
if (line1.isOn(plane1))
```

The following functions test if lines or planes are parallel, perpendicular, or coincident:

```
if (line1.isParallelTo(line2))
if (line1.isParallelTo(plane1))
if (line1.isPerpendicularTo(line2))
if (line1.isPerpendicularTo(plane1))
if (line1.isColinearTo(line2))
if (plane1.isParallelTo(plane2))
if (plane1.isPerpendicularTo(plane2))
if (plane1.isCoplanarTo(plane2))
```

The following functions return the intersections of lines and planes:

```
if (line1.intersectWith(line2,p1))
if (line1.intersectWith(plane1,p1))
if (plane1.intersectWith(plane2,line1))
```

Parametric Geometry

The following sections discuss working with parametric geometry.

Curves

Curves and surfaces in the AcGe library are parametric. A curve is the result of mapping an interval of the real line into 2D or 3D modeling space using an evaluator function with one argument, such as f(*u*). Similarly, a surface is a mapping from a 2D domain into 3D modeling space using an evaluator function based on two arguments for example, f(u, v). Each 2D and 3D curve class has a `getInterval()` function that returns the parametric interval. This function has two forms: the first returns the interval; the second returns the interval as well as the start point and endpoint of the curve.

NOTE If the interval is unbounded in either direction, the start points and endpoints do not have meaning.

Characteristics

Curves have the following characteristics:

- Orientation
- Periodicity
- Closure
- Planarity
- Length

The orientation of a curve is determined by the direction in which its parameter increases. You can use the `AcGeCurve2d::reverseParam()` or `AcGeCurve3d::reverseParam()` function to reverse the orientation of a curve.

Some curves are periodic, which means that they repeat themselves after a certain interval. For example, the period of a circle is 2π. Use these functions to determine whether a curve is periodic:

```
Adesk::Boolean
AcGeCurve2d::isPeriodic(double& period) const;

Adesk::Boolean
AcGeCurve3d::isPeriodic(double& period) const;
```

A closed curve has start points and endpoints that are the same. Curves can be either closed or open. Use these functions to determine whether a curve is closed:

```
Adesk::Boolean
AcGeCurve2d::isClosed(
    const AcGeTol&=
    AcGeContext::gTol) const;

Adesk::Boolean
AcGeCurve3d::isClosed(
    const AcGeTol&=
    AcGeContext::gTol) const;
```

A 3D curve can be planar (meaning that all of its points reside in the same plane) or nonplanar. Use this function to determine whether a 3D curve is planar:

```
Adesk::Boolean
AcGeCurve3d::isPlanar(
    AcGePlane&,
    const AcGeTol&=AcGeContext::gTol) const;
```

Given two parameter values, you can obtain the length of the curve between these two values using the following functions:

```
double
AcGeCurve2d::length(
    double fromParam, double toParam,
    double=AcGeContext::gTol.equalPoint())
    const;

double
AcGeCurve3d::length(
    double fromParam, double toParam,
    double=AcGeContext::gTol.equalPoint())
    const;
```

You can use the `AcGeCurve2d::evalPoint()` and `AcGeCurve3d::evalPoint()` functions to obtain the model space point that corresponds to a given parametric value. If your application performs evaluation frequently, you'll probably find the `AcGePointOnCurve3d` and `AcGePointOnCurve2d` classes more efficient (see "Special Evaluation Classes" on page 738). The curve functions for evaluating points are as follows:

```
AcGePoint2d
AcGeCurve2d::evalPoint(double param) const;

AcGePoint2d
AcGeCurve2d::evalPoint(
    double param, int numDeriv,
    AcGeVector2dArray& derivArray) const;

AcGePoint3d
AcGeCurve3d::evalPoint(double param) const;

AcGePoint3d
AcGeCurve3d::evalPoint(
    double param, int numDeriv,
    AcGeVector3dArray& derivArray) const;
```

Degeneracy

Certain operations can result in the creation of degenerate entities. Degenerate means that, although the resulting object belongs to a particular class, its geometry may no longer conform to the requirements of that class. For example, if you begin with a circular arc and then set its start angle equal to its end angle, you actually have a point instead of a circular arc. Geometrically, the object is a point, but its runtime type is still a circular arc. You can use one of the `isDegenerate()` functions to determine whether the object is degenerate. The first version of each pair of functions returns the type. The second version returns a nondegenerate object of a different runtime type. In the previous example, it would return a point:

```
Adesk::Boolean
AcGeCurve2d::isDegenerate(
    AcGe::EntityId& degenerateType,
    const AcGeTol&=AcGeContext::gTol)
    const;
Adesk::Boolean
AcGeCurve2d::isDegenerate(
    AcGeEntity2d*& pConvertedEntity,
    const AcGeTol&=AcGeContext::gTol)
    const;

Adesk::Boolean
AcGeCurve3d::isDegenerate(
    AcGe::EntityId& degenerateType,
    const AcGeTol&=AcGeContext::gTol)
    const;

Adesk::Boolean
AcGeCurve3d::isDegenerate(
    AcGeEntity3d*& pConvertedEntity,
    const AcGeTol&=AcGeContext::gTol)
    const;
```

Surfaces

The orientation of a surface partially determines its evaluated normal vectors. A parametric surface has two parameters, `u` and `v`, each representing the direction of the parametric lines on the surface. If you take the cross-product of the `u` tangent vector and the `v` tangent vector at the same point, you obtain a vector that is normal to the surface. This vector is the natural normal of the surface at that point. You can reverse the orientation of a surface by calling the following function:

```
AcGeSurface&
AcGeSurface::reverseNormal()
```

The surface evaluator returns either the natural normal or its inverse, depending on whether `reverseNormal()` has been called an even or odd number of times. The following function returns a value of TRUE if the orientation of the surface is opposite to the natural orientation:

```
Adesk::Boolean
AcGeSurface::isNormalReversed() const
```

This example constructs a circle and projects it onto the *XY* plane. The type of the projected entity is then checked to see what type of entity it was projected into:

```
AcGePlane     plane;   // Constructs XY-plane.
AcGePoint3d   p1(2,3,5);
AcGeVector3d  v1(1,1,1);
AcGeCircArc3d circ (p1, v1, 2.0);
AcGeEntity3d  *projectedEntity = circ.project(plane,v1);

if (projectedEntity->type() == AcGe::kEllipArc3d)
    ...
else if (projectedEntity->type() == AcGe::kCircArc3d)
    ...
else if (projectedEntity->type() == AcGe::kLineSeg3d)
    ...
```

The following example constructs a NURBS curve and finds the closest point on the curve to the point `p1`. The closest point is returned as an `AcGePointOnCurve3d` object from which the coordinates and parameter value are obtained:

```
AcGeKnotVector      knots;
AcGePoint3dArray    cntrlPnts,
AcGePointOnCurve3d  pntOnCrv;
AcGePoint3d         p1(1,3,2);

knots.append (0.0);
knots.append (0.0);
knots.append (0.0);
knots.append (0.0);
knots.append (1.0);
knots.append (1.0);
knots.append (1.0);
knots.append (1.0);
cntrlPnts.append (AcGePoint3d(0,0,0));
cntrlPnts.append (AcGePoint3d(1,1,0));
cntrlPnts.append (AcGePoint3d(2,1,0));
cntrlPnts.append (AcGePoint3d(3,0,0));
AcGeNurbCurve3d     nurb (3, knots, cntrlPnts);

nurb.getClosestPointTo(p1,pntOnCrv);
p2 = pntOnCrv.point();
double param = pntOnCrv.parameter();
```

Special Evaluation Classes

The following section describes classes in the AcGe library with which you can evaluate points on curves and surfaces. These classes are `AcGePointOnCurve2d`, `AcGePointOnCurve3d`, and `AcGePointOnSurface`.

A parametric curve is defined by a continuous function that maps some interval of the real line (possibly the entire real line) into either 2D or 3D space, depending on whether the curve is 2D or 3D. A parametric surface is defined by a continuous function that maps some connected subset of the `uv` plane (possibly the entire `uv` plane) into 3D space. The point on a parametric curve or surface that corresponds to a particular parameter value can be obtained by evaluating the function at that parameter value. For curves the parameter value is a scalar, and for surfaces the parameter value is a 2D point.

Many geometric modeling systems that support parametric curves and surfaces contain evaluator functions for computing points on parametric curves and surfaces. These evaluators typically have input arguments for the parameter value at which the curve or surface is to be evaluated and for the number of derivatives that are to be returned. They also have output arguments for the evaluated point and an array of vectors for the derivatives. Sometimes evaluators contain additional parameters for requesting and returning the normal vector at a particular parameter value.

In addition to such evaluator functions (methods called `evalPoint()`) for every curve and surface class, the AcGe library contains the evaluator classes `AcGePointOnCurve2d`, `AcGePointOnCurve3d`, and `AcGePointOnSurface`, through which the curve and surface evaluators can be accessed. The curve and surface evaluators can also be accessed through the `AcGePointOnCurve2d`, `AcGePointOnCurve3d`, and `AcGePointOnSurface` classes. These classes serve two main purposes:

- They encapsulate all the geometric information about a particular point on a curve or surface such as parameter value, model space coordinates, derivatives, and curvature.
- They provide an interface to the curve and surface evaluators that is simpler and more efficient than the traditional evaluator interface of most CAD systems.

The public interface to the AcGePointOnCurve2d, AcGePointOnCurve3d, and AcGePointOnSurface classes is identical except for minor differences in the member function names. For example, the AcGePointOnCurve3d class contains the function deriv(), which returns the derivative vector, while the AcGePointOnSurface class contains two functions, uDeriv() and vDeriv(), to return the u and v partial derivatives. The remainder of this section describes how to use the AcGePointOnSurface class, but this description applies to the AcGePointOnCurve2d and AcGePointOnCurve3d classes as well, because their interface is very similar to that of the AcGePointOnSurface class.

To use the AcGePointOnSurface class to evaluate points and derivatives, you must specify which surface is to be evaluated and the parameter value at which the evaluation is to be done. The following two member functions set the surface and parameter value of an AcGePointOnSurface object:

```
AcGePointOnSurface&  setSurface (const AcGeSurface&);
AcGePointOnSurface&  setParameter (const AcGePoint2d&);
```

After you call setSurface(), all subsequent evaluations are performed on that surface until you call setSurface() again for a different surface. Similarly, after you call setParameter(), all subsequent query functions return information pertaining to that parameter value until setParameter() is called again for a different parameter value. For example, consider if srf is an AcGeSurface object, param is an AcGePoint2d object, and pntOnSrf is an AcGePointOnSurface object, then the following code evaluates the point and first derivatives on srf at the parameter value param:

```
pntOnSrf.setSurface (srf);
pntOnSrf.setParameter (param);
AcGePoint3d    pnt3d = pntOnSrf.point();
AcGeVector3d   uFirstPartial = pntOnSrf.uDeriv(1),
               vFirstPartial = pntOnSrf.vDeriv(1);
```

In practice, you rarely, if ever, call setSurface() or setParameter() directly. Instead you call these functions indirectly through member functions of the AcGePointOnSurface class. For example, the point() function, which returns the model space point at a particular parameter value, has three different signatures:

```
AcGePoint3d    point ()  const;
AcGePoint3d    point (const AcGePoint2d& param);
AcGePoint3d    point (
    const AcGeSurface& srf,
    const AcGePoint2d& param);
```

The first signature takes no parameters and assumes that the surface and parameter value have already been set by previous calls to `setSurface()` and `setParameter()`. The second signature assumes that the surface has already been set by a previous call to `setSurface()`, but it calls `setParameter(param)` to set the parameter value before evaluating. The third signature calls `setSurface(srf)` and `setParameter(param)` to set the surface and parameter value before evaluating. Only the first member function is declared as `const`; the other two modify the object by setting the surface and/or parameter value. The direct calls to `setSurface()` and `setParameter()` can now be removed from the previous code as follows:

```
AcGePoint3d   pnt3d = pntOnSrf.point ( srf, param );
AcGeVector3d  uFirstPartial = pntOnSrf.uDeriv(1),
              vFirstPartial = pntOnSrf.vDeriv(1);
```

The first statement causes `setSurface(srf)` and `setParameter(param)` to be called before the evaluation is performed. Subsequent evaluations are performed on the same surface and at the same parameter value until `setSurface()` or `setParameter()` is called again, either directly or indirectly. Therefore, the second statement does not need to respecify either the `srf` or `param` arguments. All the evaluation functions of the `AcGePointOnSurface` class follow the same pattern of having three different signatures:

```
AcGeVector3d   uDeriv (int order)  const;
AcGeVector3d   uDeriv (int order, const AcGePoint2d& param);
AcGeVector3d   uDeriv (
    int order, const AcGeSurface& srf,
    const AcGePoint2d& param);

AcGeVector3d   vDeriv (int order)  const;
AcGeVector3d   vDeriv (int order, const AcGePoint2d& param);
AcGeVector3d   vDeriv (
    int order, const AcGeSurface& srf,
    const AcGePoint2d& param);

AcGeVector3d   mixedPartial ()  const;
AcGeVector3d   mixedPartial (const AcGePoint2d& param);
AcGeVector3d   mixedPartial (
    const AcGeSurface& srf,
    const AcGePoint2d& param);

AcGeVector3d   normal ()  const;
AcGeVector3d   normal (const AcGePoint2d& param);
AcGeVector3d   normal (
    const AcGeSurface& srf,
    const AcGePoint2d& param);
```

Similarly, there are three constructors for the `AcGePointOnSurface` class:

```
AcGePointOnSurface ();
AcGePointOnSurface (const AcGeSurface& srf);
AcGePointOnSurface (
    const AcGeSurface& srf,
    const AcGePoint2d& param);
```

When using the first constructor, you do not specify a surface or parameter value. Presumably, you set the surface and parameter value before the first evaluation. To prevent the construction of an uninitialized object, the first constructor sets the surface to `AcGePlane::kXYPlane`, which is just the *XY* plane, and sets the parameter value to the default value `(0,0)`. The second constructor calls `setSurface(srf)` and sets the parameter value to the default value of `(0,0)`. The third constructor calls `setSurface(srf)` and `setParameter(param)`. The second constructor is especially useful in functions in which a surface is passed in as an argument:

```
void func (const AcGeSurface& srf)
{
    AcGePointOnSurface  pntOnSrf (srf);
    .
    .
    .
}
```

The constructor calls `setSurface(srf)` so that all subsequent evaluations in this function are performed on `srf`.

Because the `AcGePointOnSurface` class encapsulates both the parametric and model space information about a particular point on a surface, it is useful for functions that need to return information about one or more distinct points on a surface. For instance, the `AcGeSurface` class contains the member function:

```
void getClosestPointTo (
    const AcGePoint3d& pnt3d,
    AcGePointOnSurface& closestPoint,
    const AcGeTol& =
    AcGeContext::gTol) const;
```

This function returns the closest point on the surface to the input point `pnt3d`. The closest point is returned as an `AcGePointOnSurface` object, which contains the parameter value, model space point, and other information about that particular point on the surface. All functions in the AcGe library that return an `AcGePointOnSurface` object as an output argument (non-`const`) have already called `setSurface()` and `setParameter()` for that argument. Therefore, after calling such a function, you do not need to reset the surface or parameter value. For example, the following code obtains the parameter value, model space point, and first derivatives of the closest point on the surface `srf` to the point `pnt3d`:

```
// Compute the closest point on the surface to pnt3d.
AcGePointOnSurface    closestPoint;
srf.getClosestPointTo (pnt3d, closestPoint);

// Get parameter value, model space point, and first derivative
// vectors of closest point.
AcGePoint2d   param = closestPoint.parameter();
AcGePoint3d   pnt3d = closestPoint.point();
AcGeVector3d  uFirstPartial = closestPoint.uDeriv(1),
              vFirstPartial = closestPoint.vDeriv(1);
```

None of the calls to `point()`, `uDeriv()`, or `vDeriv()` needs to specify the surface or parameter value, because they were already set by `getClosestPointTo()`. In general, `setSurface()` and `setParameter()` should not be called unless you explicitly intend to change the surface or parameter value of the `AcGePointOnSurface` object. For example, the first statement in the following code indirectly calls `setSurface()` and `setParameter()`. The second and third statements are inefficient because they make unnecessary calls to `setSurface()` and `setParameter()`, using the exact same arguments as the first statement.

```
AcGePoint3d     pnt3d = pntOnSrf.point (srf, param);
AcGeVector3d   uFirstPartial = pntOnSrf.uDeriv (1, srf, param);
AcGeVector3d   vFirstPartial = pntOnSrf.uDeriv (1, param);
```

This code executes correctly; however, it is more efficient to write it as follows:

```
AcGePoint3d     pnt3d = pntOnSrf.point (srf, param);
AcGeVector3d   uFirstPartial = pntOnSrf.uDeriv ();
AcGeVector3d   vFirstPartial = pntOnSrf.uDeriv ();
```

The AcGePointOnCurve2d, AcGePointOnCurve3d, and AcGePointOnSurface classes not only provide a way to encapsulate the parameter space and model space information of a point on a curve or surface, they also provide a simpler and more natural interface to the curve and surface evaluators than the traditional evaluators. A typical C-style surface evaluator looks something like the following:

```
void evaluate (
    int numDeriv, double u, double v, Point& pnt,
    Vector[] derivArray);
```

Here, you specify the parameter value (the parameter value of a surface is the 2D point whose coordinates are u, v) and request how many derivatives are to be returned. The evaluator then computes the point and requested derivatives at the specified parameter value. If requesting derivatives, you must know the order in which they are returned. For example, is the mixed partial stored in the fourth or fifth element of the array? You must also make sure that you do not pass in an array that is too small, or else memory overwrite will occur. This can be a problem when the evaluator is originally called for zero derivatives or one derivative (with an array size of 2 for derivArray) and is later changed to return two derivatives. If you forget to increase the size of derivArray, then memory overwrite occurs because the evaluator returns five derivative vectors (two first derivatives and three second derivatives) into an array that can only hold two vectors.

With the AcGePointOnSurface class, you request point, derivative, and normal information in a simple fashion using the point(), uDeriv(), vDeriv(), mixedPartial(), and normal() functions. The names of these functions indicate clearly which values they are returning, and there is no danger of memory overwrite. You do not have to index into an array to obtain derivative vectors and run the risk of making a mistake and using the wrong index for one or more of the vectors. The AcGePointOnSurface class provides an interface to the surface evaluator, which results in simpler code that is also more readable and understandable to other programmers.

In addition to providing a simpler and more natural interface to the curve and surface evaluators, the AcGePointOnCurve2d, AcGePointOnCurve3d, and AcGePointOnSurface classes provide a more efficient interface as well over the traditional evaluators. This is because each of these classes contains a pointer to a data area that can be used by the evaluators to store information between evaluations. For instance, the NURBS evaluator uses this area to store power basis matrices, which are not stored as part of the surface definition. By using this data area, the evaluators can avoid recomputing the same data that was computed in a previous evaluation and thus operate more efficiently. This data is not part of the curve or surface classes because evaluations might take

place in more than one area in an alternating way, which would result in inefficient loss of the local evaluation data in switching context.

This data area also allows the evaluators to be much more efficient when a transformation has been applied to the `AcGePointOnSurface` object. If the `transformBy()` function is invoked on an `AcGePointOnSurface` object, it causes subsequent evaluations to be transformed by the specified transformation without actually transforming the underlying surface. This means that the evaluators must apply the transformation to each point, derivative, and normal vector that they compute. By using the data area of the `AcGePointOnSurface` object, the evaluators can avoid having actually to apply this transformation for each evaluation. For instance, the `AcGePlane` class contains the data members `mPoint`, `mUAxis`, and `mVAxis`, which define the origin and axes of the plane. The `AcGePlane` evaluator evaluates a point with the following statement:

```
AcGePoint3d pnt3d = mPoint + param.x * mUAxis +
                    param.y * mVAxis;
```

If `transformBy()` has been called for the `AcGePointOnSurface` object, then this transformation must be applied to `pnt3d` before it is returned to the caller. The evaluator can avoid the expense of a matrix multiply by storing the transformed `mPoint`, `mUAxis`, and `mVAxis` in the `AcGePointOnSurface` data area. Then the above statement will evaluate the point in the transformed location without the extra expense of a matrix multiply. This is an especially useful ability in applications such as assembly modeling, where curves and surfaces have been transformed into assembly space by a positioning transformation.

Tips for Efficient Use of Curve and Surface Evaluators

To gain the maximum efficiency from the curve and surface evaluators, you should reuse the `AcGePointOnCurve2d`, `AcGePointOnCurve3d`, and `AcGePointOnSurface` objects as much as possible when you are performing many evaluations on the same curve or surface. For example, suppose that `func1` and `func2` both perform evaluations on the same surface `srf` and `func1` calls `func2`. Then the `AcGePointOnSurface` object that `func1` uses for evaluations should be passed to `func2`:

```
void func1 (const AcGeSurface& srf)
{
    AcGePointOnSurface  pntOnSrf (srf);
    .
    .  // Evaluate some points and derivatives.
    .
```

```
    func2 ( pntOnSrf );
    .
    .
    .
}
void func2 (AcGePointOnSurface& pntOnSrf)
{
    // Evaluate some points and derivatives using pntOnSrf
    // passed in from func1.
}
```

By passing `pntOnSrf` to `func2`, the evaluator can continue to use the same data area that was used for all the evaluations in `func1`. If `func1` does not pass the `AcGePointOnSurface` object to `func2`, then `func2` must declare a new `AcGePointOnSurface` object, which will create a new data area and recompute data that was computed in `func1`. The following code executes correctly; however, it is less efficient than the previous code:

```
void func1 (const AcGeSurface& srf)
{
    AcGePointOnSurface  pntOnSrf (srf);
    ...
    func2 (srf);
    ...
}

void func2 (const AcGeSurface& srf)
{
    AcGePointOnSurface   pntOnSrf (srf);
    .
    .   // Evaluate some points and derivatives, using new
        // pntOnSrf declared above.
    .
}
```

Reusing the same `AcGePointOnSurface` object is important for evaluator-intensive applications, such as surface-surface intersectors or finite-element mesh generators. In the case of a surface-surface intersector, the top-level function should declare two `AcGePointOnSurface` objects (one for each surface) and pass these objects down through all of the lower-level routines. In this way, the application obtains maximum use of data that is saved between evaluations and obtains the maximum efficiency from its surface evaluators.

To obtain the best use of the `AcGePointOnCurve2d`, `AcGePointOnCurve3d`, and `AcGePointOnSurface` classes, a large number of these objects should never be in scope at the same time for the same curve or surface. In most situations, only one of these objects should be in scope for a particular curve or surface.

Persistent AcGe Entities

This section describes the classes used to provide persistence for AcGe entities, and illustrates implementation of persistence. Three classes are used to provide persistence for AcGe entities: AcGeFiler, AcGeLibVersion, and AcGeFileIO.

AcGeFiler is an abstract class encapsulating the interface for the serializing requirements of AcGe. The user must provide an implementation derived from AcGeFiler. In particular, reading and writing of external entities, and all surfaces except AcGePlane, require the dwgFiler() function on AcGeFiler to be implemented.

AcGeLibVersion encapsulates the version of AcGe. It is maintained by the system. The user of AcGe keeps track of the version of AcGe being used through the global variable, AcGe::gLibVersion. All writes and reads of AcGe entities are performed in the context of the version of AcGe being used. Typically, the user must write AcGe::gLibVersion to the file before writing any other AcGe entity (correspondingly, it would be the first AcGe object read from a file in a subsequent read). The following functions are used to write and read this object (also see the following discussion of the AcGeFileIO class):

```
Acad::ErrorStatus outFields (AcGeFiler*, const AcGeLibVersion&)
Acad::ErrorStatus inFields (AcGeFiler*, AcGeLibVersion&)
```

The file I/O functions of AcGe entities are scoped within the AcGeFileIO class. These are a collection of static functions for reading and writing of AcGe entities.

AcGe Persistency Examples

The following is an example of reading and writing of an AcGeExternalSurface with a concrete subclass of AcGeFiler. For the purpose of this example, the serializing filer is a DWG filer. Consequently, it provides persistency by reading and writing DWG format:

```
#include "gefiler.h"
class AcDbDwgFiler;
class AcGeDwgFiler : public AcGeFiler
{
public:
// Construct the filer with DWG filer.
// All read and write methods are implemented
// by delegating to this filer.

    AcGeDwgFiler (AcDbDwgFiler* = NULL);
```

```
    // Read/write methods.
    //
    Acad::ErrorStatus      readBoolean(Adesk::Boolean*);
    Acad::ErrorStatus      writeBoolean(Adesk::Boolean);
    Acad::ErrorStatus      readBool(bool*);
    Acad::ErrorStatus      writeBool(bool);
    Acad::ErrorStatus      readChar(char*);
    Acad::ErrorStatus      writeChar(char);
    Acad::ErrorStatus      readShort(short*);
    Acad::ErrorStatus      writeShort(short);
    Acad::ErrorStatus      readLong(long*);
    Acad::ErrorStatus      writeLong(long);
    Acad::ErrorStatus      readUChar(unsigned char*);
    Acad::ErrorStatus      writeUChar(unsigned char);
    Acad::ErrorStatus      readUShort(unsigned short*);
    Acad::ErrorStatus      writeUShort(unsigned short);
    Acad::ErrorStatus      readULong(unsigned long*);
    Acad::ErrorStatus      writeULong(unsigned long);
    Acad::ErrorStatus      readDouble(double*);
    Acad::ErrorStatus      writeDouble(double);
    Acad::ErrorStatus      readPoint2d(AcGePoint2d*);
    Acad::ErrorStatus      writePoint2d(const AcGePoint2d&);
    Acad::ErrorStatus      readPoint3d(AcGePoint3d*);
    Acad::ErrorStatus      writePoint3d(const AcGePoint3d&);
    Acad::ErrorStatus      readVector2d(AcGeVector2d*);
    Acad::ErrorStatus      writeVector2d(const AcGeVector2d&);
    Acad::ErrorStatus      readVector3d(AcGeVector3d*);
    Acad::ErrorStatus      writeVector3d(const AcGeVector3d&);
    // Set/Get methods
    //
    AcGeDwgFiler&    setDwgFiler     (AcDbDwgFiler*);
    AcDbDwgFiler*    dwgFiler        ();

protected:
    AcDbDwgFiler*    mpFiler;
};

// Inline methods.
//
inline
AcGeDwgFiler::AcGeDwgFiler(AcDbDwgFiler* filer) : mpFiler(filer)
{}

inline AcGeDwgFiler&
AcGeDwgFiler::setDwgFiler(AcDbDwgFiler* filer)
{
    mpFiler = filer;
    return *this;
}

inline AcDbDwgFiler*
AcGeDwgFiler::dwgFiler()
{
    return mpFiler;
}
```

The next code fragment illustrates the implementation of a few functions. Other functions are implemented in the same manner:

```
Acad::ErrorStatus AcGeDwgFiler::readBoolean(Adesk::Boolean* data)
{
   return mpFiler ? mpFiler->readBoolean(data) : Acad::eNoDatabase;
}
Acad::ErrorStatus AcGeDwgFiler::writeBoolean(Adesk::Boolean data)
{
    return mpFiler ?
            mpFiler->writeBoolean(data) : Acad::eNoDatabase;
}
```

The next example illustrates a persistent class that uses `AcGeExternalSurface`. The code after the class declaration illustrates reading and writing of the external surface class. In particular, note that `AcGe::gLibVersion` is written out first and subsequently read first prior to writing or reading of the external surface class:

```
class AcGeExternalCurve2d;
class AcGeExternalCurve3d;
class AcGeExternalBoundedSurface;
class AcGeExternalSurface;

class  AcGePersistentXEnt : public AcDbObject
{
public:
    ACRX_DECLARE_MEMBERS (AcGePersistentXEnt);

    AcGePersistentXEnt ();
    ~AcGePersistentXEnt ();

    Acad::ErrorStatus dwgOutFields (AcDbDwgFiler*) const;
    Acad::ErrorStatus dwgInFields  (AcDbDwgFiler*);
    AcGeExternalSurface*           mpXSrf;
};
```

The implementation of this class is as follows:

```
ACRX_DXF_DEFINE_MEMBERS(
    AcGePersistentXEnt, AcDbObject, 0,
    GEPERSISTENT, "GeometryLib");
Acad::ErrorStatus
AcGePersistentXEnt::dwgOutFields(AcDbDwgFiler* filer) const
{
    assertReadEnabled();
    Acad::ErrorStatus stat = AcDbObject::dwgOutFields(filer);
    if (stat != Acad::eOk)
    {
        ADS_ASSERT(0);
        return stat;
    }
```

```
    // Only interested in a file filer.
    //
    if (filer->filerType() != AcDb::kFileFiler)
        return stat;
    AcGeDwgFiler   geDwgFiler(filer);
    stat = AcGeFileIO::outFields(
        &geDwgFiler,
        AcGe::gLibVersion);
    if ((stat = AcGeFileIO::outFields(&geDwgFiler,
        *mpXSrf)) != Acad::eOk)
        return stat;
    return stat;
}
Acad::ErrorStatus
AcGePersistentXEnt::dwgInFields(AcDbDwgFiler* filer)
{
    assertWriteEnabled();
    Acad::ErrorStatus stat = AcDbObject::dwgInFields(filer);
    if (stat != Acad::eOk)
    {
        ADS_ASSERT(0);
        return stat;
    }
    // Only interested in a file filer.
    //
    if (filer->filerType() != AcDb::kFileFiler)
        return stat;
    AcGeDwgFiler    geDwgFiler(filer);
    AcGeLibVersion  gelibVersion;
    if ((stat = AcGeFileIO::inFields(&geDwgFiler,
        gelibVersion)) != Acad::eOk)
        return stat;
    acutPrintf("\n... Reading External Surface\n");
    mpXSrf = new AcGeExternalSurface;
    ADS_ASSERT(mpXSrf);
    if ((stat = AcGeFileIO::inFields(&geDwgFiler,
        *mpXSrf, gelibVersion))  != Acad::eOk)
        return stat;
    return stat;
}
```

Using the Boundary Representation Library

28

In This Chapter

- Overview
- Domain
- Limitations
- Class Hierarchy
- Topological Objects
- AcBr Class Descriptions
- Enumerated Types
- Building an Application

This chapter shows how to use the AcBr library (*libacbr.dll*) to access topological, geometric, and analytic data contained in certain AutoCAD entities, such as solids, bodies, and regions (that is, objects of class `AcDb3dSolid`, `AcDbBody`, and `AcDbRegion`), and myriad derived types (for example, objects of class `AcDbPart`, `AcAsSurface`, and compatible client-defined types). For the purpose of brevity, this chapter refers to all of these objects collectively as solids.

Overview

The AcBr library can be used with the following AutoCAD entities:

- `AcDb3dSolid` represents a solid; it encloses one or more volumes.
- `AcDbRegion` represents a planar surface; it might contain multiple coplanar surfaces.
- `AcDbBody` is the concrete base class for all boundary representation objects not covered by `AcDb3dSolid` or `AcDbRegion`, including derived types defined by Autodesk Mechanical Desktop and client applications.
- `AcDbPart` represents a solid or sheet body in the context of an assembly or feature in Autodesk Mechanical Desktop.
- `AcAsSurface` represents a single surface as a sheet body in Autodesk Mechanical Desktop.

The AcBr library provides read-only access to a subset of modeling data contained in AutoCAD solids. These solids are not required to be database active, and may be created in any of the following ways:

- AutoCAD object creation commands (such as SPHERE), or equivalent AutoLISP scripts.
- Autodesk Mechanical Desktop object creation commands (such as ADREVOLVE), or equivalent AutoLISP scripts.
- Invocation of the AutoCAD EXPLODE command on a part or assembly in Autodesk Mechanical Desktop.
- File importation using OPEN, DXFIN, ACISIN, ADSATIN, VDAFSIN, STEPIN, AMIDFIN, or IGESIN.
- Programmatic instantiation of primitives using `AcDb3dSolid::createFrustum()`, `AcDb3dSolid::createBox()`, `AcDb3dSolid::createWedge()`, `AcDb3dSolid::createSphere()`, `AcDb3dSolid::createTorus()`, `AcDbRegion::createFromCurves()`.

Typical uses of the AcBr library include the following:

- Transferring entity or subentity data into your application for display, analysis, or manipulation.
- Locating particular features of interest in the solid and querying for associated data, such as geometry.
- Transferring entity data to another modeling system (that is, data exchange).
- Meshing the surface data in the solid for display, analysis, or manipulation.
- Supporting analysis (such as point and line containment, bounding blocks, and mass properties).

Domain

AutoCAD solids are boundary representations (often referred to as B-rep models), consisting of a collection of topological connectivity objects and associated geometric boundary objects. The topological objects are defined in the AcBr library and are described later in this chapter, whereas the geometric objects are defined in the AcGe library.

Objects defined or generated by the AcBr library reside in three-dimensional Euclidean model space (E^3). The only exceptions are geometric objects defined in the two-dimensional parameter space of a surface (such as parameter curves and parameter points).

In general, only the 2-manifold topological domain is supported by the AcBr library. Singularities (which are geometric degeneracies) are supported in order to represent the apex of a cone, but wire bodies and mixed dimensionality solids (which may include dangling wires and faces) are not supported; nor can they be realized in AutoCAD.

The general nonmanifold domain is a superset of the 2-manifold domain, and allows distinct solid volumes to touch at single points, curves, or faces; and allows any combination of wireframe, sheet, and solid objects. The following nonmanifold objects are supported by AutoCAD and the AcBr library:

- Two 2-manifold solids united along a shared edge or vertex
- An `AcDbBody` object containing a single face

A topological object may be unbounded (that is, it may have no lower dimensional bounding topology) only in the following cases:

- A closed surface, which is intrinsically bounded in both the `u` and `v` directions (such as a full torus or sphere), is represented by a face that has no loop boundaries.
- A closed curve, which is intrinsically bounded (such as a full circle or ellipse), is represented by an edge that has coincident start and end vertices.

Limitations

Certain operations cannot support nonuniform scaling. This includes all functions that return an external curve or surface (including NURBS surfaces).

The entire chain of transforms from the subentity path is cached at the time that an `AcBr` object's subentity path is set (for efficiency reasons). If a block reference is moved, it will point to a new transform matrix but the AcBr object will not know that its cached transform is out of date. If an insert is changed to refer to a different AutoCAD entity, the subentity path simply no longer has relevance and should be updated to reflect the new entity reference before being used to reinitialize all relevant AcBr objects.

Singularities (such as the apex of a cone) map to edges in AutoCAD and thus can be used to initialize an `AcBrEdge` for the express purpose of querying for the vertex, but cannot be queried for curve geometry or used to set an `AcBrLoopEdgeTraverser`. They can also be accessed using an `AcBrLoopVertexTraverser`, as a singularity corresponds to a single loop boundary of a face.

Just as with `AcDbObject` pointers, AcBr objects cannot be used once the AutoCAD database object has been closed in the database or goes out of scope; they are not persistent. Any change to the database object will be flagged as an `eBrepChanged` error, unless the validation level has been set to ignore database changes. An out-of-scope or closed database object will generally cause `Acad::eNotInDatabase` to be returned.

Class Hierarchy

The AcBr class hierarchy is a subset of the ObjectARX class hierarchy, and defines the following classes:

```
AcRxObject
    ├─ AcBrEntity
    │   ├─ AcBrBrep
    │   ├─ AcBrComplex
    │   ├─ AcBrEdge
    │   ├─ AcBrFace
    │   ├─ AcBrLoop
    │   ├─ AcBrShell
    │   └─ AcBrVertex
    ├─ AcBrMeshControl
    │   └─ AcBrMesh2dControl
    ├─ AcBrMeshEntity
    │   ├─ AcBrElement
    │   │   └─ AcBrElement2d
    │   └─ AcBrNode
    ├─ AcBrTraverser
    │   ├─ AcBrBrepComplexTraverser
    │   ├─ AcBrBrepEdgeTraverser
    │   ├─ AcBrBrepFaceTraverser
    │   ├─ AcBrBrepShellTraverser
    │   ├─ AcBrBrepVertexTraverser
    │   ├─ AcBrBrepShellTraverser
    │   ├─ AcBrBrepVertexTraverser
    │   ├─ AcBrComplexTraverser
    │   ├─ AcBrEdgeLoopTraverser
    │   ├─ AcBrElement2dNodeTraverser
    │   ├─ AcBrFaceLoopTraverser
    │   ├─ AcBrLoopEdgeTraverser
    │   ├─ AcBrLoopVertexTraverser
    │   ├─ AcBrMesh2dElement2dTraverser
    │   ├─ AcBrShellFaceTraverser
    │   ├─ AcBrVertexEdgeTraverser
    │   └─ AcBrVertexLoopTraverser
    ├─ AcBrHit
    ├─ AcBrHitPath
    └─ AcBrMesh2dFilter
```

Note that AcBr objects are not derived from `AcDbObject`, and therefore cannot be registered with the AutoCAD database.

Topological Objects

Topological objects are either primary or secondary, depending on whether they are bound to a specific topological dimension.

Primary topological objects are used to cover an evaluated model space completely. They are defined in terms of point sets and are also referred to as n-simplexes, where `n` is their topological dimension. A 0-simplex is a vertex, a 1-simplex is an edge, a 2-simplex is a face, and a 3-simplex is a complex. They do not include their boundaries, but they can be bounded by simplexes of any lower dimension.

The primary topological objects are the following:

Complex	A connected topologically three-dimensional region of points R^3 in E^3. It is a volume constructed out of vertices, faces, and edges. A complex is usually bounded by one or more shells.
Face	A connected topologically two-dimensional region of points R^2 in E^3. It is a bounded, orientable subset of a surface on a shell boundary of a complex. A face is usually bounded by one or more loops.
Edge	A connected topologically one-dimensional region of points R^1 in E^3. It is a bounded, orientable subset of a curve on a loop boundary of a face. An edge is usually bounded by one or two vertices.
Vertex	A connected topologically zero-dimensional region of points R^0 in E^3. It is a single point on a face. A vertex is bounded only by itself.

The geometry returned by each of these primary topological objects can be queried further using the Autodesk Geometry Library.

Secondary topological objects are connected collections of primary topological objects, and are not necessarily bound to a specific topological dimension. They represent the boundary mapping from a higher-dimension simplex to a set of lower-dimension simplexes that define a connected part of its boundary. Each primary topological object belongs to at least one secondary topological object.

The secondary topological objects are the following:

Brep	A collection of everything in an evaluated space; that is, a collection of all of the primary and other associated secondary topological objects for a unique E^3. At the very least, this collection must contain a single complex.
Shell	An unordered collection of faces that bound a complex. At the very least, this collection must contain a single face. There may be at most one exterior shell, and there must be an exterior shell for there to be interior shells (voids).
Loop	An ordered collection of edges and vertices that form the connected boundaries of a face, which may consist of a single vertex (for a singularity, such as the apex of a cone) or an ordered connected sequence of edges. There may be at most one exterior loop, and there must be an exterior loop for there to be interior loops (holes).

Using Topological Objects in Your Program

Proper use of the AcBr library involves interactions between several internal and external objects: `AcBrEntity`, `AcBrTraverser`, and their derived classes; `AcDbFullSubentityPath` and its components from the AcDb library (`AcDbObjectId`, `AcDbSubentId`, and so on); and geometry objects from the AcGe library (`AcGeSurface`, `AcGeCurve3d`, `AcGePoint3d`, and so on).

In a boundary representation model, topological objects are used as the "glue" that holds the model together, and geometric data is referenced by the topological objects. To access the details of a boundary representation model's shape, you may either traverse the topological objects (using a traverser) or proceed directly to a topological object of interest (using the `AcDbFullSubentPath` corresponding to a screen pick).

Using Solid Objects

Create an `AcBrBrep` object, and initialize it with the `set()` function, using an `AcDbFullSubentPath` that was previously set using `kNullSubentId` and the owning AutoCAD object ID. Create an `AcBrBrepFaceTraverser` to gain access to the faces of the entire solid, and initialize with `setBrep()`. For each position of the traverser's topological adjacency list (using repeated calls to the `next()` function), `getFace()` provides an `AcBrFace` that can be used to access the geometry. Other traversers provide access to hierarchically lower topological objects and their associated geometry.

Using Specific Subentities

Create an appropriate AcBr object (`AcBrFace`, `AcBrEdge`, or `AcBrVertex`), and initialize it with the `set()` function, using an `AcDbFullSubentPath` that was previously set using the subentity and the owning AutoCAD object ID.

Using the Geometry of a Face, Edge, or Vertex

Call the `get*` functions on an appropriate and initialized AcBr object to create the associated `AcGeSurface`, `AcGeCurve3d`, or `AcGePoint3d`.

Using the Mesh Data of a Brep, Complex, Shell, or Face

Create an `AcBrMesh2dFilter`, using an appropriate AcBr object and an `AcBrMesh2dControl` object, and call `generate()` on an instantiated `AcBrMesh2d` object to generate a surface mesh. This mesh can be traversed to access nodal geometry and other data.

Using Topological Traversers in Your Program

Traversers are used to walk through the topological structure of the solid. They are either global or hierarchical.

Global Searches

Global traversers (such as `AcBrBrepFaceTraverser` and `AcBrBrepEdgeTraverser`) provide the ability to traverse all of the topological objects in the solid (complexes, shells, faces, edges, vertices).

Hierarchical (Local) Searches

Hierarchical traversers provide the ability to walk through local areas of the solid, either top-down or bottom-up. Several of these traversers, such as `AcBrLoopEdgeTraverser` and `AcBrLoopVertexTraverser`, also provide queries to distinguish between multiple uses of the same edge or vertex.

For example, an edge has no inherent knowledge of which face it is being referenced from. So, the parameter curve data, which depends on the underlying surface on which the edge is defined, is queried from `AcBrLoopEdgeTraverser`.

Downward hierarchical traversers give access to the topologically adjacent subentities that make up the boundary of a given topological object (that is, the traverser list owner).

From Topological Traversers to Objects

Topological traversers represent adjacency lists of topological objects that are connected in the context of a higher-dimension topological object.

Each particular type of traverser exposes both the object it is using for context (that is, adjacency list owner) and the object it is currently pointing to (that is, adjacency list position) with `get*` and `set*` functions.

Topological traversers

Class	Objects
AcBrBrepComplexTraverser	AcBrBrep (owner) AcBrComplex (position)
AcBrBrepShellTraverser	AcBrBrep (owner) AcBrShell (position)
AcBrBrepFaceTraverser	AcBrBrep (owner) AcBrFace (position)
AcBrBrepEdgeTraverser	AcBrBrep (owner) AcBrEdge (position)
AcBrBrepVertexTraverser	AcBrBrep (owner) AcBrVertex (position)
AcBrShellFaceTraverser	AcBrShell (owner) AcBrFace (position)
AcBrFaceLoopTraverser	AcBrFace (owner) AcBrLoop (position)
AcBrLoopEdgeTraverser	AcBrLoop (owner) AcBrEdge (position)
AcBrLoopVertexTraverser	AcBrLoop (owner) AcBrVertex (position)
AcBrVertexLoopTraverser	AcBrVertex (owner) AcBrLoop (position)
AcBrVertexEdgeTraverser	AcBrVertex (owner) AcBrEdge (position)
AcBrEdgeLoopTraverser	AcBrEdge (owner) AcBrLoop (position)

Vertices for edges

An edge can have at most two vertices. These are exposed by explicit functions in the `AcBrEdge` class (see the next section), as a traverser would be wasteful for such a trivial adjacency.

Vertices for loops

A loop can have many vertices, but may have as few as one (in the case of a single edge, or in the case of singularity, where there is no edge geometry, such as the apex of a cone). The LoopVertex traversal covers both the general list of vertex boundaries on a face as well as singularities. This list may be more economical than dumping the edges on a loop, if the only thing of interest is the point geometry for the face boundary.

EdgeLoop traversal

This class defines the functions that are related to the radial ordering of faces that share a common edge. In order to provide the tightest coupling with edge list traversals (`AcBrLoopEdgeTraverser`), the face is represented by its loop boundary at the shared edge.

The `setEdgeAndLoop()` function sets the edge owner and the loop starting position. The `setEdge()` function sets the edge owner and the loop starting position. The loop position cannot be set separately from the edge, as radial traversals should be tightly coupled with face-contextual edge lists (that is, `AcBrLoopEdgeTraverser`).

VertexLoop traversal

This class defines the functions that are related to the radial ordering of faces that share a common vertex. To provide the tightest coupling with edge list traversals (`AcBrLoopEdgeTraverser`), the face is represented by its loop boundary at the shared vertex.

The `setVertexAndLoop()` function sets the vertex owner and the loop starting position. The `setVertex()` function sets the vertex owner and sets the loop starting position. The loop position cannot be set separately from the vertex, as radial traversals should be tightly coupled with face-contextual vertex lists (that is, `AcBrLoopVertexTraverser`).

From Mesh Traversers to Mesh Objects

Mesh traversers represent adjacency lists of mesh objects that are connected in the context of a higher-dimension mesh object.

Each particular type of traverser exposes both the object it is using for context (that is, adjacency list owner) and the object it is currently pointing to (that is, adjacency list position) with `get*` and `set*` functions.

Mesh traversers	
Class	**Objects**
AcBrMesh2dElement2dTraverser	AcBrMesh2d (owner) AcBrElement2d (position)
AcBrElement2dNodeTraverser	AcBrElement2d (owner) AcBrNode (position)

MeshElement traversal — This class defines the functions that are pertinent to a 2D element. It is used to seed a downward hierarchical traversal of a 2D mesh, or to traverse all of the unique 2D elements and nodes in a 2D mesh.

ElementNode traversal — This class defines the functions that are pertinent to a node in the context of a 2D element. It is used to get access to node data and the geometry of the original surface, such as surface normals and UV parameter pairs. Nodes are used by more than one mesh element and may be associated with more than one surface since there is node sharing at the common boundaries of the original surfaces.

AcBr Class Descriptions

The AcBr library is composed of several functional components. All functions involving `AcBr` objects pass them as references. Consequently, an instance of the `AcBrFace` class must have been created before calling a `getFace()` function from a traverser or loop.

Entity Classes

Boundary representation objects are typically built using a default AcBr constructor and then initialized either with a `set()` function or with a traverser and one of its `get*` functions.

All AcBr classes support copy constructors; assignment operators; `isEqualTo()`, `isNull()`, and `set()` and `get()` semantics; and other functions and queries.

The entity classes include the following:

- `AcBrEntity`
- `AcBrBrep`
- `AcBrComplex`
- `AcBrShell`
- `AcBrFace`
- `AcBrLoop`
- `AcBrEdge`
- `AcBrVertex`

Containment Classes

Containment objects are never built directly by the user. They are returned by line containment queries on entities derived from `AcBrEntity`.

The `AcBrHit` class is a containment class.

Mesh Classes

Mesh objects are never built directly by the user, except where noted in the *ObjectARX Reference*. They are returned by mesh traversal queries.

The mesh classes include the following:

- `AcBrMeshEntity`
- `AcBrMesh`
- `AcBrMesh2d`
- `AcBrElement`
- `AcBrElement2d`
- `AcBrNode`
- `AcBrMesh2dFilter`
- `AcBrMeshControl`
- `AcBrMesh2dControl`

Traverser Classes

Traverser objects are typically built using a default `AcBrTraverser*` constructor and then initializing with one of the `set*` functions. Note that the list owner must be set before the list position can be set independently, to provide context.

All classes derived from `AcBrTraverser` support copy constructors, assignment operators, `isEqualTo()`, and `isNull()`, along with general traversal functions.

The initializer functions are semantically bound to the AcBr types appropriate to the specific traverser (that is, the two types contained in the derived traverser class name, such as `AcBrBrep` and `AcBrEdge` for `AcBrBrepEdgeTraverser`).

All initializer functions reset the criteria for `next()` and `done()`. They fall into the general algorithmic categories as follows:

- `setListOwnerAndCurrentPosition` from another traverser, using its list owner as current position and its current position as list owner (that is, swap the list owner and current position). This algorithm is only valid for mapping between associated traversers such as `AcBrLoopEdgeTraverser` and `AcBrEdgeLoopTraverser`.
- `setListOwnerAndCurrentPosition` from an AcBr object, using it as the current position and its owner as the list owner. This algorithm is only valid in cases where the list owner is unambiguous, such as the shell owner of a face on setting `AcBrShellFaceTraverser`.
- `setListOwner` from another traverser, using its current position as list owner and defaulting the current position to the first position in the new adjacency list. This algorithm is only valid for setting downward hierarchical traversers using another downward hierarchical traverser from the next level up (such as using an `AcBrShellFaceTraverser` to initialize an `AcBrFaceLoopTraverser`), or for setting upward hierarchical traversers using another upward hierarchical traverser from the next level down (such as using an `AcBrVertexEdgeTraverser` to initialize an `AcBrEdgeLoopTraverser`).
- `setListOwner` from an AcBr object, using it as the list owner and defaulting the current position to the first position in the new adjacency list. This algorithm is valid for all traverser types.
- `setCurrentPosition` from an AcBr object, using it as the current position in an already established list. This algorithm is valid for most traverser types but requires that the list owner has already been set previously.

The traverser classes include the following:

- `AcBrTraverser`
- `AcBrBrepComplexTraverser`
- `AcBrBrepShellTraverser`
- `AcBrBrepFaceTraverser`
- `AcBrBrepEdgeTraverser`
- `AcBrBrepVertexTraverser`
- `AcBrComplexShellTraverser`

- `AcBrShellFaceTraverser`
- `AcBrFaceLoopTraverser`
- `AcBrLoopEdgeTraverser`
- `AcBrLoopVertexTraverser`
- `AcBrVertexLoopTraverser`
- `AcBrVertexEdgeTraverser`
- `AcBrEdgeLoopTraverser`
- `AcBrMesh2dElement2dTraverser`
- `AcBrElement2dNodeTraverser`

Enumerated Types

The `AcBr` struct contains enumerated types that are unique to the AcBr library and that are used as return codes or on the argument list of local class functions. The various enum fields are described below.

Error Return Codes

The `AcBr::ErrorStatus` enum is returned by all non-Boolean valued functions. The error code may be either a global AutoCAD error code (as listed in the `Acad::ErrorStatus` enum), cast to the local `AcBr::ErrorStatus` enum, or a local boundary representation library error code. The local error codes are defined using a starting base of 3000, so as not to conflict with the AutoCAD, AutoCAD Mechanical API, or Autodesk Mechanical Desktop error codes.

Validation Level

The `AcBr::ValidationLevel` enum sets the level of validation for an AcBr object. If `kFullValidation` (the default upon object instantiation) is specified, every function that accesses the brep topology (directly or indirectly) first checks the associated AutoCAD database object to see if it has been modified since the AcBr object was last set. This is an expensive operation, but it guarantees all brep data is within scope. If `kNoValidation` is specified, the database is not checked unless it is critical to the completion of the function's tasks. This is more efficient but does not guarantee the brep data is within scope. Setting the validation level on an object-by-object basis prevents any collisions between applications loaded simultaneously.

ShellType

The `AcBr::ShellType` enum classifies a shell as interior, exterior, and so on. Peripheral shells are returned as `kShellExterior`, and there is only one such shell per brep or region (by industry convention). Voids are returned as `kShellInterior`, and there may be several per brep or complex (providing there is also an exterior shell).

LoopType

The `AcBr::LoopType` enum classifies a loop as interior, exterior, and so on. Peripheral loops are `kLoopExterior`, and there is only one such loop per face (by industry convention). Holes are returned as `kLoopInterior`, and there may be several per face (providing there is also an exterior loop). Cones and cylinders (whether with an elliptical or circular base) have at least two base loops (if they are complete in both `u` and `v`), which are returned as `kLoopWinding` as opposed to `kLoopExterior` due to the restriction of there being just one exterior loop (along with the fact that neither base loop is a hole). Singularities (such as the apex of a cone) are returned as `kLoopUnclassified`. All loops on spheric and toric surfaces, as well as closed periodic NURBS, return `kLoopUnclassified`.

Mesh Element Shape Control

The `AcBr::Element2dShape` enum controls the shape of elements generated by a 2d mesh. It is only relevant to the mesh filter. A 1d mesh (corresponding to the discretization of the boundary curves of the original face) can be emulated by specifying `kAllPolygons`.

Building an Application

An application that uses the AcBr library must have the library file *libacbr.dll* available to link against.

More importantly, the library is needed to ensure proper registration of the AcBr classes with ObjectARX at runtime.

Therefore it is important that *libacbr.dll* be explicitly loaded by the application, if it has not already been loaded by the modeler or another application. The best way to ensure this is to use `acrxDynamicLoader()` and `acrxClassDictionary()` to load *libacbr.dll* and to check if it has been loaded.

The following code fragment provides an example:

```
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        if (!acrxClassDictionary->at("AcBrEntity")) {
            acrxDynamicLinker->loadModule("libacbr.dll", 1);
            acutPrintf("\n libacbr loaded \n");
            }
        acedRegCmds->addCommand(
            "MY_APP",
            "MY_CMD",
            "MY_CMD",
            ACRX_CMD_MODAL,
            &myCmdImp);
        acrxUnlockApplication(pkt); // try to allow unloading
        break;
    case AcRx::kUnloadAppMsg:
        acedRegCmds->removeGroup("MY_APP");
        break;
    default:
        break;
    }
    return AcRx::kRetOK;
}
```

NOTE It is important not to unload *libacbr.dll* upon exit from an application, as other applications (or the modeler) may still depend on its presence.

Sample Application Using the AcBr Library

A sample ObjectARX application using the boundary representation library is included in the ObjectARX SDK in the *objectarx\utils\brep\samples\brepsamp* directory. This directory includes a *Readme* file that describes the sample application and how to use it.

The sample application is based on an interactive approach, which may not be the best approach for all applications. Objects are picked using selection sets, resbufs, and GS markers, and are transferred to the AcBr library using `AcDbFullSubentPaths`.

Part VII

Appendixes

Migrating ADS Programs to ObjectARX

In This Appendix

A

- Migrating to ObjectARX
- Loading Applications: ADS versus ObjectARX
- Building ADS Applications in the ObjectARX Program Environment
- Sample ObjectARX Application
- ObjectARX-Exclusive Data Type

To simplify the migration of AutoCAD Development System applications to the ObjectARX program environment, the ADS library was ported to the ObjectARX program environment. The ObjectARX version is almost identical to the ADS version. This appendix provides a comparison of how programs are loaded in both ADS and ObjectARX and includes a sample program that has been ported from ADS to ObjectARX.

Migrating to ObjectARX

Existing ADS applications must be ported to ObjectARX. All of the library functions that were previously available in the ADS library are included in ObjectARX. Applications that frequently communicated with AutoCAD through the ADS library or other calls run faster in the ObjectARX program environment than in the ADS program environment.

The acrxEntryPoint() Function

An ObjectARX application does not have a main, because it is a DLL. Also, the application does not call `ads_init()`, `ads_abort()`, and `ads_link()`. The ADS programmer implements a function `acrxEntryPoint()` with the following signature:

```
extern "C"
AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* appId);
```

The first parameter, a data member of the `AcRx` class called `msg`, represents the message sent from the ObjectARX kernel to the application.

The second parameter is an opaque handle to data passed to the lock and unlock functions for the application. The function returns a status code, such as `RSRSLT` or `RSERR`.

AutoCAD calls into the ObjectARX module `acrxEntryPoint()` to pass messages to the application. All requests to invoke functions through `acedDefun()` are made by `acrxEntryPoint()`, simplifying the migration of ADS programs to the ObjectARX program environment.

ObjectARX applications respond to the value of the AcRx class's data member `AppMsgCode` rather than to the return value of `ads_link()`, `ads_exit()`, and `ads_abort()`.

Header Files

The following header files must be included in the ObjectARX application source file:

adesk.h	Contains standard definitions for the ObjectARX program environment.
rxdefs.h	Establishes an ObjectARX application and interacts with AutoCAD through `acrxEntryPoint()`.
adslib.h	Establishes platform-specific definitions and includes *adscodes.h* and *ads.h*.
adscodes.h	Contains definitions of code values that are returned by (or passed to) ADS library functions.
ads.h	Contains the ADS library type definitions and function declarations.

The following header files can be included in ObjectARX application source files:

adsdlg.h	Contains Dialog Control Language-related declarations for creating dialog boxes.
ol_errno.h	Contains symbolic codes for the error values used by the AutoCAD system variable ERRNO.
adsdef.h	Establishes definitions for the ADS program environment.

The *adslib.h* header file contains directives for including *adscodes.h*, *adsdef.h*, and *ads.h*; therefore, an application source file needs to contain only the following directive:

```
#include "adslib.h"
```

An ObjectARX application does not need to include *ol_errno.h* unless it uses the symbolic codes defined there to handle the value of ERRNO. The application doesn't need to include *adsdlg.h* unless it creates dialog boxes.

Loading Applications: ADS versus ObjectARX

Application loading in the ObjectARX program environment is more flexible than in the ADS program environment. In the ADS program environment, the ADS application always stays in memory.

In both the ObjectARX and ADS program environments, an application can be loaded automatically when AutoCAD is invoked, if the application name is listed in *acad.rx* or *acad.ads,* respectively.

In the ObjectARX program environment, certain applications, such as Render, can be invoked when one of its functions is called. This capability conserves memory for large applications like Render, because it loads the application only for a short time during the drawing session.

ObjectARX and ADS application loading are different in the following ways:

- When a drawing is closed or when another drawing is opened, ObjectARX applications are not unloaded. An ObjectARX application is unloaded when AutoCAD is terminated or when the application has no dependents and `acedArxUnload()` or an equivalent function is called.
- In the ObjectARX program environment, a drawing is present when either the `kLoadDwg` or `kUnloadDwg` messages or both are received. These messages are received in the event loop when either an ADS or ObjectARX application is initialized or unloaded.
- In the ADS program environment, a drawing is present when a `kInitAppMsg` or `kUnloadAppMsg` is received.

NOTE In the ObjectARX program environment, do not assume that a drawing is present when a `kInitAppMsg` or `kUnloadAppMsg` is received.

- In the ObjectARX program environment, call `(arxload)` or `acedArxLoad()` to load applications. In the ADS program environment, call `(xload)` or `ads_xload()` to load applications.
- ObjectARX applications listed in *acad.rx* are loaded automatically when AutoCAD is invoked, and unlike ADS applications, the ObjectARX applications are initialized before a drawing is present and before the ADS or Visual LISP environments are initialized. In the ADS program environment, the counterpart to *acad.rx* is *acad.ads*. Put the names of the application modules in the appropriate file, one name per line.

Building ADS Applications in the ObjectARX Program Environment

ADS applications are implemented under ObjectARX as DLLs and have a default file extension of *.arx*. For instructions on building ADS applications in the ObjectARX program environment, see the AutoCAD *readdev.hlp* file.

The procedure described in the AutoCAD *readdev.hlp* file for debugging ADS applications in the ADS program environment also applies to the ObjectARX program environment.

Sample ObjectARX Application

Compare the sample applications of the factorial programs distributed with AutoCAD. The *fact.ccp* program is for the ObjectARX program environment, while *fact.c* is for the ADS program environment. The following version of the *fact.cpp* declares and implements `acrxEntryPoint()`; otherwise, *fact.cpp* is almost identical to *fact.c*. The function `acrxEntryPoint()` replaces the main in *fact.c*:

```
#include  <stdio.h>
#include "adslib.h"
#include  "rxdefs.h"


// Utility definition to get an array's element count (at compile
// time).   For example:
//
//     int  arr[] = {1,2,3,4,5};
//     ...
//     printf("%d", ELEMENTS(arr));
//
// would print a five. ELEMENTS("abc") can also be used to tell
// how many bytes are in a string constant INCLUDING THE TRAILING
// NULL.

#define ELEMENTS(array) (sizeof(array)/sizeof((array)[0]))


// All the functions that we'll define will be listed in a single
// table, together with the internal function that we call to handle
// each. The functions all take a single argument (the resbuf that
// has the arguments) and return an integer (RTNORM or RTERROR for
// good or bad status).
```

```
// First, define the structure of the table: a string giving the
// AutoCAD name of the function, and a pointer to a function
// returning type int.

struct func_entry { char *func_name;
    int (*func) _((struct resbuf *));
};

// Here we declare the functions that handle the calls; at the moment
// there are two of them.

int fact       _((struct resbuf *rb));
int squareroot _((struct resbuf *rb));

// Here we define the array of function names and handlers.

static struct func_entry func_table[] = {
    {/*MSG0*/"fact", fact},
    {/*MSG0*/"sqr", squareroot},
};

// To add more functions to this table, just put them in the list,
// after declaring the function names.  Note that in standard C it's
// all right to have a superfluous comma after the last item.


// The code from here to the end of dofun() is UNCHANGED when you
// add or delete functions.


// Declarations of other local functions:

void     main       _((int, char **));
int      dofun      _((void));
int      funcload   _((void));
ads_real rfact      _((int x));
ads_real rsqr       _((ads_real x));

// ACRXENTRYPOINT -- This function replaces main() for an ObjectARX
// program.

extern "C" AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* ptr)
{
    switch(msg) {
    case AcRx::kInitAppMsg:
        acrxUnlockApplication(ptr);
        break;
    case AcRx::kInvkSubrMsg:
        dofun();
        break;
    case AcRx::kLoadADSMsg:
        funcload();
    }
    return AcRx::kRetOK;
}
```

```
// FUNCLOAD  --  Define this application's external functions.
//               Return RTERROR on error, else RTNORM.

static int funcload()
{
    int i;

    for (i = 0; i < ELEMENTS(func_table); i++) {
        if (!acedDefun(func_table[i].func_name, (short)i))
            return RTERROR;
    }
    return RTNORM;
}

// DOFUN -- Execute external function (called upon an RQSUBR
//        request). Return value from the function executed, RTNORM
//          or RTERROR.

static int dofun()
{
    struct resbuf *rb;
    int val;

    // Get the function code and check that it's within range.
    // (It can't fail to be, but paranoia doesn't hurt.)
   if ((val = acedGetFunCode()) < 0 || val >= ELEMENTS(func_table))
    {
        acdbFail(/*MSG2*/"Received nonexistent function code.");
        return RTERROR;
    }

    // Fetch the arguments, if any.
    rb = acedGetArgs();

    // Call the handler and return its success-failure status.
    val = (*func_table[val].func)(rb);
    acutRelRb(rb);
    return val;
}


// The code from the beginning of main() to here is UNCHANGED when
// you add or delete functions.
```

```
// FACT -- First set up the argument, then call the factorial
//         function.
static int
fact(struct resbuf *rb)
{
    int x;

    if (rb == NULL)
        return RTERROR;

    if (rb->restype == RTSHORT) {
        x = rb->resval.rint;          // Save in local variable
    } else {
        acdbFail(/*MSG3*/"Argument should be an integer.");
        return RTERROR;
    }

    if (x < 0) {                      // Check argument range
        acdbFail(/*MSG4*/"Argument should be positive.");
        return RTERROR;
   } else if (x > 170) {           // Avoid floating-point overflow
        acdbFail(/*MSG5*/"Argument should be 170 or less.");
        return RTERROR;
    }

   acedRetReal(rfact(x));          // Call the function itself, and
                                   // return the value to AutoLISP
    return RTNORM;
}

// This is the implementation of the actual external factorial
// function.

static ads_real rfact(int n)
{
    ads_real ans = 1.0;

    while (n)
        ans *= n--;

    return ans;
}

// SQUAREROOT -- First set up the argument, then call the root
// function.
```

```
static int
squareroot(struct resbuf *rb)
{
    ads_real x;

    if (rb == NULL)
        return RTERROR;               // A proper error msg would
                                      // be better.

    if (rb->restype == RTSHORT) {     // Save in local variable.
        x = (ads_real) rb->resval.rint;
    } else if (rb->restype == RTREAL) {
        x = rb->resval.rreal;         // Can accept either real
                                      // or integer.
    } else {
        acdbFail(
            /*MSG6*/
            "Argument should be a real or integer value.");
        return RTERROR;
    }

    if (x < 0) {                      // Check argument range.
        acdbFail(/*MSG7*/"Argument should be positive.");
        return RTERROR;
    }

   acedRetReal(rsqr(x));           // Call the function itself, and
                                   // return the value to AutoLISP.

    return RTNORM;
}

// This is the implementation of the actual external function

static ads_real rsqr(ads_real x)       // Square root by Newton's
                                       // method.
{
    int n = 50;
    ads_real y, c, cl;

    if (x == 0.0) {
        return 0.0;
    }

    y = (x * 2 + .1) / (x + 1.0);
    c = (y - x / y) / 2;
    cl= 0.0;

    while ((c != cl) && (n-- > 0)) {
        y -= c;
        cl = c;
        c  = (y - x / y) / 2;
    }
    return y;
}
```

ObjectARX-Exclusive Data Type

`ACAD_GRAPHICS` is a new data type defined at runtime for extended entity data in the ObjectARX program environment. The data is in the form of binary chunks. The DXF sequence and contents of this data define a graphics image of the entity for a DXF file loaded into AutoCAD without its defining application. For more information, see the *AutoCAD Customization Guide*, appendix D, "Drawing Interchange File Formats."

Programmable Dialog Boxes

In This Appendix

B

- Overview
- Function Sequence Outline
- Definitions and Declarations
- Handling Tiles

ObjectARX contains a set of functions that is collectively called the Programmable Dialog Box (PDB) package. PDB functions define dialog box controls, functionality, and linkage to the application.

This chapter describes how to use the PDB. See the *ObjectARX Reference* for a synopsis and catalog of the functions referred to in this chapter. For dialog box design guidelines, see the *AutoCAD Customization Guide*. For information about using Microsoft Foundation Class, chapter 8, “MFC Topics.”

Overview

Dialog box programming involves two phases:

- *Designing the dialog box* Dialog boxes are defined by text files written in dialog control language (DCL). The DCL description of a dialog box defines how the box appears and what it contains. For more information, see Part III, "Programmable Dialog Box Reference," in the *AutoCAD Customization Guide*.
- *Supporting the dialog box in your application* The parts of a dialog box define how it behaves; however, the use and behavior of a dialog box depend on the application that employs it.

Function Sequence Outline

Dialog boxes are for interactive use. A script can start a dialog box but cannot control it or provide input once it is opened. This is the same as the `acedCommand()` and `acedCmd()` functions.

The examples given in this section demonstrate the typical dialog box function sequence as follows:

1. Load the DCL file with an `ads_load_dialog()` call.
2. Call `ads_new_dialog()` to display a particular dialog box on the AutoCAD graphics screen.

   Check the status of the value that `ads_new_dialog()` returns. Calling `ads_start_dialog()` when the `ads_new_dialog()` call has failed can have unpredictable results.
3. Initialize the dialog box by setting up tile values, lists, and images.

   The functions typically called at this time are as follows:

   - `ads_set_tile()` and `ads_mode_tile()` for general tile values and states
   - `ads_start_list()`, `ads_add_list()`, and `ads_end_list()` for list boxes
   - `ads_dimensions_tile()` for setting tile values, along with the following image creation functions:

     ```
     ads_start_image()
     ads_vector_image()
     ads_fill_image()
     ads_slide_image()
     ads_end_image()
     ```

You can also call `ads_client_data_tile()` at this time to associate application-specific data with the dialog box and its components.

Call `ads_action_tile()` at this point to set up callback functions.

4 Call `ads_start_dialog()` to turn control over to the dialog box so that the user can enter input.

5 Process user input from within your functions. This is when you are most likely to use `ads_get_tile()`, `ads_get_attr()`, `ads_get_attr_string()`, `ads_set_tile()`, and `ads_mode_tile()`.

6 The user presses an exit button, causing a function to call `ads_done_dialog()`, which then causes `ads_start_dialog()` to return a value. At this point, unload the DCL file by calling `ads_unload_dialog()`.

This sequence can be shown schematically in pseudocode as follows:

```
load_dialog

    new_dialog
        action_tile ;and other initializations
    start_dialog
        ;Then from within the action expressions / callback
        ;functions:
        get_tile ;and other input handling
        set_tile
        done_dialog

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 `ads_load_dialog` call then loads all dialog boxes at once, and you can call `ads_new_dialog` for any dialog box. If memory is limited, however, you may have to create multiple DCL files and use `ads_unload_dialog` to remove one set of dialog boxes from memory before you load another set with `ads_load_dialog`.

Example Dialog Box

The following sections show how to create and display a sample dialog box. The DCL file defines a dialog labeled “Hello, world” that contains a text tile and a single OK button.

Sample DCL File

The following DCL for creating the sample dialog box is saved in a file called *hello.dcl*:

```
hello : dialog {
    label = "Sample Dialog Box";
    : text {
        label = "Hello, world";
    }
    ok_only;
}
```

Sample ObjectARX Function

The following ObjectARX function displays the sample:

```
int showalert()
{
    int dcl_id, dbstatus;
    ads_hdlg dbhello;

    // Load the DCL file.
    //
    ads_load_dialog("hello.dcl", &dcl_id);
    // Initialize the dialog.
    //
    if (ads_new_dialog("hello", dcl_id, NULLCB, &dbhello)
        != RTNORM) {
        acdbFail("Unable to initialize dialog box called
            \"hello\"\n");
        return BAD; // Exit if this does not work
    }
    // Associate an action. End expression with a key, in this
    // case, the OK button. End the dialog when OK is pressed.
    //
    ads_action_tile(dbhello, "accept", accept_OK);

    // Display the dialog box.
    //
    ads_start_dialog(dbhello, &dbstatus);
    ads_unload_dialog(dcl_id); // Unload the DCL file
    return GOOD;
}

static void CALLB accept_OK(ads_callback_packet *cpkt)
{
    // DLGOK == User pressed OK
    //
    ads_done_dialog(cpkt->dialog, DLGOK);
}
```

After the `ads_start_dialog()` call, the dialog box remains active until the user selects a tile (usually a button). The function `ads_action_tile()` is called, and it calls a callback function. The arguments to a callback function

are enclosed in a single callback packet structure. The callback calls `ads_done_dialog()`.

A complex dialog box requires more calls to `ads_action_tile()`, possibly other initialization calls, and probably more input handling between the `ads_start_dialog()` and `ads_unload_dialog()` calls. The overall calling sequence, though, remains the same.

Functions Not Allowed While a Dialog Box Is Active

While a dialog box is active, the `ads_start_dialog()` function is executing its instructions. Under these conditions, you cannot call certain ObjectARX functions, because they either affect the display, which must not change while the dialog box is visible, or they require user input that does not involve the dialog box.

If the application calls one of the these functions before it calls `ads_done_dialog()`, AutoCAD terminates all dialog boxes and displays the following error message:

```
AutoCAD rejected function
```

The AutoCAD CMDACTIVE system variable has a bit that indicates whether a dialog box is active. For more information on system variables, see the *AutoCAD Command Reference*.

If the user must enter input based on the graphics screen rather than use the dialog box itself (for example, to specify a point or an entity), you must hide the dialog box. That is, you must call `ads_done_dialog()` to redisplay the graphics screen, and then restart the dialog box after the user has made the selection. The following lists show the functions that are not allowed.

AutoCAD Queries and Command Functions

The following AutoCAD functions cannot be called while a dialog box is active:

- `acedCommand()`
- `acedCmd()`
- `acedHelp()`
- `acedOsnap()`

User Input Functions

The following user input functions cannot be called while a dialog box is active:

- `acedGetInt()`
- `acedGetReal()`
- `acedGetString()`
- `acedGetPoint()`
- `acedGetCorner()`
- `acedGetDist()`
- `acedGetAngle()`
- `acedGetOrient()`
- `acedGetKword()`
- `acedGetInput()`
- `acedDragGen()`

Display Control Functions

The following display control functions cannot be called while a dialog box is active:

- `acedPrompt()`
- `acedMenuCmd()`
- `acedRedraw()`
- `acedGraphScr()`
- `acedTextScr()`
- `acedTextPage()`

The functions that write text, such as `acutPrintf()`, are useful for displaying information while testing a dialog box, but they should not be used in a finished product.

Low-Level Graphics Functions

The following graphics functions cannot be called while a dialog box is active:

- `acedGrVecs()`
- `acedGrDrag()`
- `acedGrRead()`
- `acedGrText()`
- `acedGrDraw()`
- `acedGrText()`

Selection Set Functions

Interactive `acedSSGet()` calls are not allowed while a dialog box is active, but other options are allowed.

Entity-Handling Functions

The following entity-handling functions cannot be called while a dialog box is active:

- `acdbEntMod()`
- `acdbEntMake()`
- `acdbEntDel()`
- `acedEntSel()`
- `acedNEntSel()`
- `acedNEntSelP()`
- `acdbEntUpd()`

Callback Functions

To define what action is taken when a dialog box tile is selected, associate an ObjectARX function with that tile by calling the `ads_action_tile()`. Within the callback, you often need access to attributes in the DCL file. The `ads_get_tile()` and `ads_get_attr()` functions provide this access (`ads_get_attr()` gets the value saved in DCL, while `ads_get_tile()` gets the current runtime value), but the values you are most likely to use, those associated with the selected tile, are provided automatically.

In most cases, every active tile within a dialog box generates a callback. The callback function should do validity checking for its associated tile and 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 save the settings the user finally selected. Update the variables associated with tile values within the callback for the OK button, not within 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 chooses Cancel. 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 these tiles with a single callback function. The principle of not committing to the user's changes until the user specifies OK still applies. A callback function common to several tiles can be table driven, using user-defined attributes to provide values specific to each tile.

Default Actions

There is another way to define actions in addition to calling `ads_action_tile()`. You can define a `default` action for the entire dialog box when you call `ads_new_dialog()`. A tile can have only a single action at a time. If the application specifies more than one action, they supersede each other in the following order of priority:

1. The default action specified by the `ads_new_dialog()` call (used only if no action is explicitly assigned to the tile).
2. The action assigned by the last `ads_action_tile()` call. When a tile is named in more than one `ads_action_tile()` call, only the last such call (prior to `ads_start_dialog()`) has any effect (this is similar to assigning multiple values to the same variable). The PDB facility allows only one action per tile.

Passing Arguments in Callback Functions

In a callback function, the value and data of a selected tile are passed as a callback packet argument, as shown in the definition of the callback function of the earlier example:

```
static void CALLB accept_OK(ads_callback_packet *cpkt)
{
    // DLGOK == User pressed OK.
    //
    ads_done_dialog(cpkt->dialog, DLGOK);
}
```

The `CALLB` symbol that appears before the function name is defined as blank. It is simply a marker to make it easier for you to locate callback functions when you maintain your program. However, you should always use it in case it is defined differently in a future release.

The preceding example simply closes the dialog box, using only one of the arguments in the packet. A callback packet data type is defined by the following statement:

```
typedef struct {
    ads_hdlg dialog;
    ads_htile tile;
    char *value;
    void *client_data;
    int reason;
    long x, y;
} ads_callback_packet;
```

The arguments passed in the packet have the following purposes:

dialog	The handle of the dialog box.
tile	The handle of the selected tile. Instead of passing the `key` of the selected tile, the PDB package passes the callback function a tile handle (of the type `ads_htile`). You use the handle to retrieve the tile's attributes, including its key, by calling the function `ads_get_attr_string()`. (The `ads_get_attr_string()` function does not have an AutoLISP counterpart.)
value	A string that contains the value of the selected tile. The space for this string is managed by AutoCAD; treat it as read-only. If you need to change the value of the tile, use `ads_set_tile()`. If the tile is a list box (or a pop-up list) and no item is selected, the `value` string is empty (`""`).
client_data	A pointer to the application-specific data that was initialized by `ads_client_data_tile()`. If there is no client data, this is `NULL`.
reason	The reason for the callback. This depends on what action the user took. Its value is set for any action, but you need to inspect it only when the action is associated with an `edit_box`, `list_box`, `image_button`, or `slider` tile.
x, y	When the user chooses an image button, these are set to the (*X*,*Y*) coordinates selected. The coordinates are tile coordinates within the range that `ads_dimensions_tile()` would return for the image button.

For example, to retrieve the key of the selected tile, the callback function could include the following code:

```
char newtile[TILE_STR_LIMIT];
ads_get_attr_string(cpkt->tile, "key", newtile, TILE_STR_LIMIT);
```

When retrieving a string value, be sure to allocate space for the string. This example specifies the string length by using the constant `TILE_STR_LIMIT`.

The `ads_get_attr_string()` function can retrieve other attribute values in the same way that this example retrieves the key.

Callback Reasons

The callback reason, returned as the `reason` field of a callback packet, specifies why the action or callback 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 values

Code	Symbol	Description
1	CBR_SELECT	The user selected the tile. This is the value for most action tiles.
2	CBR_LOST_FOCUS	For edit boxes, the user moved to another tile but did not make 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.
3	CBR_DRAG	For sliders, the user changed the slider value by dragging the indicator (or equivalent) but did not make a final selection. The application should not update the value of the associated variable but should update the text that displays the slider's status.
4	CBR_DOUBLE_CLICK	This callback reason always follows a CBR_SELECT. It usually means "Commit to the previous selection." It should not undo the previous selection; this can confuse and annoy the user. For list boxes or image buttons, the user double-clicked to make a final selection. For image buttons, the user double-clicked on the image button.

The meaning of a double-click on a list box or image button is up to 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 `list_box` tile's `is_default` attribute should be `true`). If the list box is not the primary tile in the dialog box, then a double-click should be treated the same as making a selection (1, or `CBR_SELECT`). List boxes that allow the user to select multiple items (`multiple_select = true;`) cannot support double-clicking.

If the main purpose of the dialog box is to select an image button, a single-click should select the button, but sometimes it is better for a single-click (or a keyboard move) to highlight the button and an ENTER or a double-click to select it.

An example of single-click image-button handling is the AutoCAD Choose Hatch Pattern dialog box (called from the BHATCH command). An example of double-click image buttons is the AutoCAD Select Text Font dialog box (called from the Set Style option on the Text submenu of the default Draw pull-down menu), which shows a list box with text style names and image buttons with equivalent text style icons. In this dialog box, a single-click on either an image button or a list item highlights both the text style name and icon, and a double-click on either makes a selection.

Nesting Dialog Boxes

You create and manage nested dialog boxes simply by calling `ads_new_dialog()` and `ads_start_dialog()` from within a callback function. 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 you should not nest dialog boxes deeper than three or four.

Although `ads_term_dialog()` terminates all dialog boxes at once, it does not return a status code, so there is no way for your application to distinguish between hiding a nested box and canceling boxes because of an error condition.

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 `ads_done_dialog()`, except that your callback function must use the `ads_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 `ads_start_dialog()` function returns the application-defined `status` when the dialog box disappears. Your program must then examine the status returned by `ads_start_dialog()` to determine the next action.

The following sample program has a button, Select Point, that hides the dialog box so that the user can specify a point on the graphics screen. The action of selecting this button causes the dialog box to end with a special status of 4:

```
ads_real x_pt, y_pt, z_pt;
ads_point pick_pt;
ads_hdlg hdlg;
int what_next;

static void CALLB
pick_callback(ads_callback_packet *cpkt)
{
    ads_done_dialog(cpkt->dialog, 4);
}

void
bmake_handler()
    // Load dialog box and do global initialization
    //
    while (what_next >= DLGSTATUS) {
    // Indicates custom return code
    // Other initialization such as ads_new_dialog(),
    // ads_action_tile(), ads_set_tile(), and
    // ads_start_list() calls.
    //
        ads_start_dialog(hdlg, &what_next);
        switch (what_next) {
        case 4:
            acedGetPoint(NULL, "Insertion base point: ",
            pick_pt);
            acdbRToS(pick_pt[X], 2, 4, x_pt);
            acdbRToS(pick_pt[Y], 2, 4, y_pt);
            acdbRToS(pick_pt[Z], 2, 4, z_pt);
            break;
        ...
        }
    }
}
```

The following example hides multiple dialog boxes:

```
// Global variables
//
ads_point pick_pt;

// These must be global because the subdlg_handler()
// function needs to be able to access them as well
// as the main dialog function.
//
ads_hdlg mdlg;
int dcl_id, what_next;
```

```
static void CALLB
hide_handler(ads_callback_packet *cpkt)
{
   ads_done_dialog(cpkt->dialog, 3);
}


static void CALLB
subdlg_handler(ads_callback_packet *cpkt)
{
    // REMEMBER: This function must never reference anything in
    // the cpkt packet because none of its fields are valid when
    // it is called explicitly in the main dialog function.
    //
    ads_hdlg sdlg;
    ads_new_dialog("subdlg", dcl_id, NULLCB, &sdlg)
    ads_action_tile(sdlg, "hide_all", hide_handler);
    ads_start_dialog(sdlg, &what_next1);
    if (what_next1 == 3) // Nested hide is in progress. */
        ads_done_dialog(mdlg, DLGSTATUS); // Hide main dialog box.
}


void
maindlg_handler()
{
    int what_next;
    ads_callback_packet dummy_pkt;

// dummy_pkt is used when this section of code explicitly calls the
// subdlg_handler() function. The subdlg_handler() function expects
// a single parameter that is a pointer to an ads_callback_packet.
// Normally a callback function is called by AutoCAD, and AutoCAD
// provides a filled-in packet, but in this code we need to call
// the callback function explicitly in order to redisplay the
// subdialog after a hide. In order to do this we need a dummy
// ads_callback_packet. It doesn't have to be filled in because
// none of its fields is ever used.
//
    ads_load_dialog("maindlg.dcl", &dcl_id);
    what_next = what_next1 = 5; // could be set to anything > 1.

    while (what_next >= DLGSTATUS) { //DLGSTATUS == 2.
        ads_new_dialog("maindlg", dcl_id, NULLCB, &mdlg)
        ads_action_tile(mdlg, "x", subdlg_handler);
        if (what_next1 == 3) {
            // This is only true on returning from a nested hide.
            // Since we are returning from a nested hide,restart the
            // subdialog.
            // Note that the main dialog has NOT been started yet.
            // It is just a bit map painted on screen (it needs an
            // ads_start_dialog() for interactivity).
            //
```

```
            subdlg_handler(&dummy_pkt);
            if (what_next1 != 3) {
                // OK or CANCEL pressed to exit the subdialog
                // so it is time to activate the main dialog that
                // was painted but not started.
                //
                ads_start_dialog(mdlg, &what_next);
            }
        } else {
            // this is executed only once upon startup of this whole
            // dialog code.
            //
            ads_start_dialog(mdlg, &what_next);
        }
        if (what_next == DLGSTATUS) { /* DLGSTATUS == 2 */
            // This if condition is true when a nested hide is
            // in progress and both dialogs are hidden.
            //
            ads_getpoint(NULL, "\nPick a point: ", pick_pt);
        }
    }
    ads_unload_dialog(dcl_id);
}
```

Definitions and Declarations

To write applications that call functions in the PDB package, include the following preprocessor directive in your program:

```
#include "adsdlg.h"
```

In addition to function declarations, *adsdlg.h* defines a number of symbols and types to use with the dialog box functions described in the *ObjectARX Reference*.

Handles for Dialog Boxes and Tiles

The *adsdlg.h* file defines two handle types:

- `ads_hdlg` identifies dialog boxes. The `ads_new_dialog()` function assigns the new dialog box an `ads_hdlg` handle for identifying the dialog box in most subsequent calls to PDB functions until `ads_done_dialog()` is called.
- `ads_htile` identifies a selected tile within callback functions.

If `ads_new_dialog()` fails, it sets the handle to `NULL` or `BAD_DIALOG`.

Callback Function Definitions

To register callback functions with `ads_action_tile()` and `ads_new_dialog()`, *adsdlg.h* defines the type `CLIENTFUNC`, which points to a callback function as shown in the following sample:

```
typedef void (*CLIENTFUNC)(ads_callback_packet *cpkt);
```

The (blank) symbol `CALLB` is defined to make callback functions easier to locate in the source code, as in the following example:

```
static void CALLB
dbox_handler(ads_callback_packet *cpkt)
```

You can use the `ads_new_dialog()` function also to specify a default callback function for the dialog box. If you don't use this feature, pass the null function pointer `NULLCB`, which is defined as follows:

```
#define NULLCB ((CLIENTFUNC) 0)
```

Status Codes

The `ads_start_dialog()` function has a `status` argument that it sets to indicate how the dialog box ended. The values for this status are shown in the following table:

Status code values

Symbol	Description
DLGOK	The user chose the OK button or its equivalent.
DLGCANCEL	The user chose Cancel or its equivalent.
DLGALLDONE	No dialog box is active; ads_term_dialog was called.
DLGSTATUS	If status is greater than or equal to DLGSTATUS, it is an application-defined status code.

The reason code passed in a callback packet (`cpkt->reason`) is an integer that indicates why the callback occurred (that is, what user action generated the callback). The callback reason codes are shown in the following table:

Callback reason code values

Symbol	Description
CBR_SELECT	The user selected the tile.
CBR_LOST_FOCUS	For edit boxes, the user moved to another tile but did not make a final selection.
CBR_DRAG	For sliders, the user changed the value by dragging the indicator (or equivalent) but did not make a final selection.
CBR_DOUBLE_CLICK	For list boxes or image buttons, the user double-clicked to make a final selection.

The symbols described in this section are used with the `ads_mode_tile()` and `ads_start_list()` functions.

The function `ads_start_list()` begins handling a list for a list box or a pop-up list. The symbols to use are shown in the following table:

List function code values

Symbol	Description
LIST_CHANGE	Change selected list contents.
LIST_APPEND	Append new list entry.
LIST_NEW	Delete old list and create new list.

The function `ads_mode_tile()` controls the focus of a tile and determines whether it is enabled. The symbols and their descriptions are shown in the following table.

Tile mode code values

Symbol	Description
MODE_ENABLE	Enable tile
MODE_DISABLE	Disable tile
MODE_SETFOCUS	Set focus to tile
MODE_SETSEL	Select edit box contents
MODE_FLIP	Flip image highlighting on or off

Attribute names and values are passed as strings; your programs need to allocate space for them. The upper limit on strings used with dialog boxes is defined in `TILE_STR_LIMIT` as 255 plus one for the null terminator, `EOS`).

Handling Tiles

Your program has some control over the tiles in the current dialog box at initialization time and action (callback) time. This section introduces the tile-handling functions.

Initializing Modes and Values

Initializing a tile includes making it the initial keyboard focus of the dialog box, disabling or enabling it, or (if it is an edit box or image) highlighting its contents. These operations are performed by `ads_mode_tile()` calls. You can also set the value of a tile with `ads_set_tile()`.

The following code displays a default value, in this case a surname, in an edit box and sets the dialog box's initial focus to that box:

```
strcpy(name_str, "Nugent"); // Default name.

ads_set_tile(hdlg, "lastname", name_str);
ads_mode_tile(hdlg, "lastname", MODE_SETFOCUS);
```

The following statement calls `ads_mode_tile()` again to highlight the edit box contents so that the user can immediately type over the default contents:

```
ads_mode_tile(hdlg, "lastname", MODE_SETSEL);
```

On some platforms, setting the focus to an edit box automatically highlights it, making this additional step unnecessary.

Changing Callback Modes and Values

At callback time, you can check the value of a tile. If the application calls for it, you can use `ads_set_tile()` again to modify this value. During callbacks, you can also use `ads_mode_tile()` to change the status of a tile. The following table shows the values of the `ads_mode_tile()` mode argument:

Tile mode values

Value	Symbol	Description
0	MODE_ENABLE	Enable tile
1	MODE_DISABLE	Disable tile
2	MODE_SETFOCUS	Set focus to tile
3	MODE_SETSEL	Select edit box contents
4	MODE_FLIP	Flip image highlighting on or off

When you use `ads_mode_tile()` to disable a tile that has the current focus, you must call `ads_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.

An 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 presses Next on the next-to-last page, the button is disabled. The same happens after pressing Previous on the second page. In both cases, the code must disable the button that was pressed, and then set the focus to a different tile.

The following example controls a cluster called "group". When the toggle is set to Off, the tiles in the cluster are inactive and should not be modified:

```
static void CALLB group_on_off(ads_callback_packet *cbpkt)
{
    ads_hdlg hdlg = cbpkt->dialog;
    char value[TILE_STR_LIMIT];

    strcpy(value, cbpkt->value);
    if (strcmp(value, "0") == 0) { // Cluster is disabled.
        ads_mode_tile(hdlg, "group", MODE_DISABLE);
    } else { // The value must equal "1".
        ads_mode_tile(hdlg, "group", MODE_ENABLE);
    }
}
```

You can inspect other attributes besides a tile's value with the `get_attr()` function. The following example retrieves the label of a button called "pressme":

```
char label_str[TILE_STR_LIMIT];
ads_get_attr(hdlg, "pressme", "label", label_str, TILE_STR_LIMIT);
```

If you use `ads_get_attr()` to retrieve a `value` attribute, it gets the value attribute saved in the DCL file (the initial value of the tile). The `ads_get_tile()` function, however, gets the current runtime value of the tile. The two values are not necessarily the same.

The `ads_get_attr()` function returns the attribute's value in a string argument (`value`). Because this function sets the value of the string, you must allocate space for it, as shown in the preceding example.

Setting Up List Boxes and Pop-Up Lists

You set up the lists displayed in list boxes and in pop-up lists by using a sequence of calls to three functions: `ads_start_list()`, `ads_add_list()`, and `ads_end_list()`. Once a list has been created, you can revise it. There are three possible operations, which are specified by the `ads_start_list()` function's operation argument (whose values are shown in parentheses in the following list).

- Create a new list (`LIST_NEW`).

  After the `ads_start_list()` call, you can call `ads_add_list()` repeatedly. Each `ads_add_list()` call adds a new item to the list. End list handling by calling `ads_end_list()`.

- Change an item in the list (`LIST_CHANGE`).

  After `ads_start_list()`, call `ads_add_list()` once to replace the item whose index was specified in the `ads_start_list()` call. (If you call

ads_add_list() more than once, it replaces the same item again.) End list handling by calling ads_end_list().

- Append an item to the list (LIST_APPEND).

  After ads_start_list(), call ads_add_list() to append an item to the end of the list. If you continue to call ads_add_list(), more items are appended until you call ads_end_list().

Regardless of which list operation you are doing, you must call the three functions in the correct sequence: ads_start_list(), then ads_add_list() (possibly more than once), then ads_end_list().

Lists are most easily represented by linked result buffers, as shown in the following example:

```
struct resbuf *appnames, *rb;

// Initialize the appnames list here.
//
...
rb = appnames;

ads_start_list(hdlg, "selections", LIST_NEW, 0);

while (rb != NULL) {
    ads_add_list(rb->resval.rstring);
    rb = rb->rbnext;
}
ads_end_list();
```

For short lists, it is easier to pass individual strings. A result-buffer list is not required.

The value of a list_box tile is the index (or indexes) of the selected item (or items). If your program needs to know the actual text associated with an index, it must save the original list. It must also track changes made by the methods shown in the following examples.

Appending list items is similar to creating a new list. For example, appnames has 12 items in it, and you want to append another list called newnames:

```
(start_list "selections" 2)
(mapcar 'add_list newnames)
(end_list)
```

In ObjectARX, you must specify an index value, but `ads_add_list()` disregards it in an append operation.

```
struct resbuf *appnames, *newnames, *rb;

rb = newnames;

ads_start_list(hdlg, "selections", LIST_APPEND, 0);
while (rb != NULL) {
    ads_add_list(rb->resval.rstring);
    rb = rb->rbnext;
}
ads_end_list();
```

Changing a single item requires only one `ads_add_list()` call. In the following example, you specify the index of the item to change:

```
ads_start_list(hdlg, "selections", LIST_CHANGE, 5);
ads_add_list("SURPRISE!");
ads_end_list();
```

You cannot delete a list item or insert an item without rebuilding the entire list.

Handling List Values

The value of a `list_box` tile can contain leading spaces. If you retrieve multiple items, do not test the value as a string comparison. Convert it to an integer first, using `atoi()`.

Assuming that a list accepts only a single selection, the following code fragment checks whether the third list item has been selected. (Be certain that the string is empty first, because `atoi()` returns 0 for an empty string as well as for the string "`0`".) The following example uses the value passed in the callback packet:

```
if (*cpkt->value != EOS) {
    if ((atoi(cpkt->value) == 2) {
    // Process the third entry.
    ...
    }
}
```

The value of a pop-up list never has a leading space, so you do not need to convert the value. Pop-up lists do not allow 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 example requires that you include the standard C library header file *string.h*. In addition, you should call the `mk_list()` function with the list box's cur-

rent value, `cpkt->value`, and a pointer to the original list. For the sake of simplicity, error messages are left out.

```
resbuf
*mk_list(char *local, struct resbuf *oldlist)
{
    char spaceset[] = {' ', '\t', '\n'}, item[TILE_STR_LIMIT];
    int nitem, i;
    struct resbuf *findrb; *usrlist, *scratch, *usrlast;
    usrlist = usrlast = NULL;

    while (item = strtok(local, spaceset) != NULL) {
        nitem = atoi(item);
        findrb = oldlist;
        for (i=0; i<nitem; i++) {
            findrb=findrb->rbnext;
        }
        if (usrlist == NULL) /* First item */
            if (scratch = acutNewRb(RTSTR) == NULL)
            // Assume it's a string.
                return NULL;
            if ((scratch->resval.rstring =
                malloc(strlen(findrb->resval.rstring))) == NULL)
                return NULL;
            strcpy(scratch->resval.rstring,
                findrb->resval.rstring);
            usrlist = usrlast = scratch;
        } else { /* Trailing items */
            if (scratch = acutNewRb(findrb->restype) == NULL)
                return NULL;
            if ((scratch->resval.rstring =
                malloc(strlen(findrb->resval.rstring))) ==
                NULL)
                return NULL;
            strcpy(scratch->resval.rstring,
                findrb->resval.rstring);
            usrlast->rbnext = scratch;
            usrlast = scratch;
        }
    }
    return usrlist;
}
```

This example also works for the degenerate case of a single selection.

Creating Images

The calling sequence to create images for image tiles and image buttons is similar to the list-handling sequence. The `ads_start_image()` function begins the creation of an image, and `ads_end_image()` ends it. However, options for what to draw are specified by the following separate function calls instead of arguments:

- `ads_vector_image()` draws a vector (a single, straight line) in the current image.
- `ads_fill_image()` draws a filled rectangle in the current image.
- `ads_slide_image()` draws an AutoCAD slide in the image.

Vectors and filled rectangles are useful for simple images, such as the color swatches (filled rectangles) 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 them, keep them simple.

The image-drawing function, `ads_vector_image()`, requires that you specify absolute coordinates, while `ads_fill_image()` and `ads_slide_image()` require a starting coordinate 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 package provides a function, `ads_dimensions_tile()`, that returns the width and height of a particular tile. Call this function before you begin creating an image. The origin of a tile, (0,0), is always its upper-left corner.

Colors can be specified as AutoCAD color numbers or as one of the "logical" color numbers shown in the following table.

ADI color numbers

Color Number	ADI Mnemonic	Description
-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

The values and mnemonics are defined by the Autodesk Device Interface (ADI).

In the following example, "`cur_color`" is an image tile to be filled entirely with a patch of red. Only one call is needed to get the image's dimensions:

```
short width, height;

ads_dimensions_tile(hdlg, "cur_color", &width, &height);
ads_start_image(hdlg, "cur_color");
ads_fill_image(0, 0, width, height, 1); // 1 == red.
ads_end_image();
```

The image-drawing functions can be used with each other. Here, the code fills an image and then draws a vertical stripe over it:

```
short width, height, x;

ads_dimensions_tile(hdlg, "stripe", &width, &height);
ads_start_image(hdlg, "stripe");
ads_fill_image(0, 0, 0, height, 3); // 3 == AutoCAD green.

// Center the vector vertically.
//
x = width/2;
ads_vector_image(x, 0, x, height, 4); // 4 == cyan.
ads_end_image();
```

The slides you display with `ads_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 *.sld* extension (for example, "`frntview`"). If the slide is in a slide library, you specify the name of the library first (without the extension), followed by the name of the slide itself (also without the extension) enclosed in parentheses (for example, "`allviews(frntview)`"). The `ads_slide_image()` function searches for the slide or slide library file according to the current AutoCAD library search path.

The slide in the following example is in a single file called *topview.sld*:

```
short x, y;

ads_dimensions_tile(hdlg, "view", &x, &y);
ads_start_image(hdlg, "view");
ads_slide_image(0, 0, x, y, "topview");
ads_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 `ads_slide_image()` call with an `ads_fill_image()` call.)

Image Button Input

You can handle an image button simply as a button—that is, to trigger a single action. However, the PDB facility also gives you the option of defining regions of the button so that the action taken depends on which 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 chose. 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 locations on the image button tile by implicitly defining image regions.

In the following example, the image button has two color swatches created by `ads_fill_image()`. You can select one or the other, depending on which region the user chooses. If the image button is divided horizontally (dark above, light below), the callback needs to test only the one dimension. Do not use a point structure of type `ads_point` to store the pickpoint coordinates, because they are passed as `long` integers:

```
char result[31]; // Global char array

static void CALLB
pick_shade(ads_callback_packet cbpkt)
{
    long threshold, pick_y = cbpkt->y;
    ads_hdlg hdlg = cbpkt->dialog;
    short x, y;

    ads_dimensions_tile(hdlg, "image_sel", &x, &y);
    threshold = y/2;

    // Remember the origin at upper left.
    //
    if (pick_y > threshold)
        strcpy(result, "Light");
    else
        strcpy(result, "Dark");
}
```

Handling Radio Clusters

Radio buttons appear in radio clusters. The value of each individual 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 package manages the values of radio buttons in a cluster, and ensures that only one is on at a time. You can assign an action to each individual 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 of the radio buttons was chosen.

In the following example, a radio cluster controls which view of a three-dimensional object is displayed after the user leaves the dialog box. This cluster contains four radio buttons (although there could be more).

```
ads_action_tile(hdlg, "view_sel", pick_view);

...

static void CALLB
pick_view(ads_callback_packet *cbpkt)
{
    char value[TILE_STR_LIMIT];
    strcpy(value, cbpkt->value);
    if (strcmp(value, "front") == 0)
        show_which = 0;
    else if (strcmp(value, "top") == 0)
        show_which = 1;
    else if (strcmp(value, "left") == 0)
        show_which = 2;
    else if (strcmp(value, "right") == 0)
        show_which = 3;
}
```

The preceding examples show each radio button associated with a single variable that takes multiple values. They may also cause additional actions, such as disabling selections in your dialog box. If the radio cluster is a large one, it is convenient to store the associated values in a table. If you use a table, structure it so that it does not depend on the order of the buttons within the cluster. The PDB package does not impose this restriction, and the order can change if the DCL 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.

Although you are not required to check the reason code, it is recommended that you do so to reduce processing. The frequency of callbacks that sliders generate depends on the platform, but some platforms generate a `CBR_DRAG` callback for every mouse movement the slider detects.

The following function shows the basic scheme of a slider-handling function. 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 the user the choice of either manipulating the slider or typing its value directly. If the user types the value in `slider_info`, the edit box callback should (conversely) update the value of the slider:

```
static void CALLB
slider_action(ads_callback_packet *cbpkt)
{
    ads_hdlg hdlg = cbpkt->dialog;
    int reason = cbpkt->reason;
    char interim[TILE_STR_LIMIT];

    // Save the interim result.
    //
    strcpy(interim, cbpkt->value);/

    // Display the result.
    //
    ads_set_tile(hdlg, "slider_info", interim);
}

static void CALLB
ebox_action(ads_callback_packet *cbpkt)
{
    ads_hdlg hdlg = cbpkt->dialog;
    int reason = cbpkt->reason;
    char interim[TILE_STR_LIMIT];

    // Save the interim result.
    //
    strcpy(interim, cbpkt->value);

    // Display the result.
    //
    ads_set_tile(hdlg, "myslider", interim);
}
```

Handling Edit Boxes

Actions and callbacks to handle edit boxes are similar to those for sliders. However, since characters in edit boxes are already visible, there is no need for action on interim results.

The following sample code checks the value but does not redisplay it:

```
static void CALLB
edit_action(ads_callback_packet *cbpkt)
{
    int reason = cbpkt->reason;

    if ((reason == CBR_LOST_FOCUS) || (reason == CBR_SELECT)) {
    // Check range, syntax, etc. on transient value.
    //
    ...
    }
}
```

To show the alternative, this example checks for `CBR_LOST_FOCUS` rather than `CBR_DOUBLE_CLICK`.

Application-Specific Data

The `ads_client_data_tile()` function assigns application-specific data to a tile. The data is available at callback time as the callback packet's `client_data` field. 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.)

The client data can be of whatever type you choose. It is declared as a pointer to `void`.

You can express client data as a pointer to a data structure associated with the dialog box or tile. The structure can be declared as temporary data local to any function at or above the level of the function that calls `ads_start_dialog()`. This lets you avoid declaring client data as global static data.

Client data is a useful way to pass information to a callback function, because no extra parameters can be added to the function.

Because your program must maintain the list displayed by a list box (or a pop-up list), client data is good for handling this information. The previous example of the `mk_list()` function already makes the list value an argument.

It is a pointer type, which lends itself to use as client data. For example, the main function to handle the same dialog box could include the following code (error checking is minimized for simplicity):

```
struct resbuf *usrhead;

int
handler()
{
    struct resbuf *csyshead, *usrhead;
    ads_hdlg cldlg;

    if (ads_new_dialog("clistdlg", dcl_id, NULLCB, &cldlg))
        return BAD;

    csyshead = acutBuildList(RTSTR, "Red-Green-Blue",
                        RTSTR, "Cyan-Magenta-Yellow",
                        RTSTR, "Hue-Saturation-Value", 0);
    if (csyshead == NULL)
        return BAD;
    ads_client_data_tile(cldlg, "colorsyslist", csyshead);
    ads_action_tile(cldlg, "colorsyslist", listcallback);
    ...

    // Start dialog box and do other processing.
    //
    ...
}
```

Next, the callback function `listcallback()` invokes `mk_list()` as follows:

```
static void CALLB
listcallback(ads_callback_packet *cpkt)
{
    if ((cpkt->reason == CBR_SELECT) ||
        (cpkt->reason == CBR_DOUBLE_CLICK)) {
        // This is not the default tile, so treat both the same.
        //
        usrhead = mk_list(cpkt->value, cpkt->client_data);
    }
}
```

Index

Symbols

.arx file extension, 657
.dbx file extension, 657
3D geometry, 730

A

abortDeepClone() function, 504
abortInsert() function, 507
abortTransaction() function, 455
abortWblock() function, 505
AC_DECLARE_EXTENSION_MODULE macro, 171
AC_IMPLEMENT_EXTENSION_MODULE macro, 171, 187
acad.lib library, 10
acad.unt file, 269
ACAD_PROXY_ENTITY type, 389
ACAD_PROXY_OBJECT type, 389
AcApDocManager class, 417
    setDefaultFormatForSave() function, 62
AcApDocManagerReactor class, 418
AcApDocument class, 61, 417
    formatForSave() function, 62
AcApDocumentIterator class, 417
AcApLayoutManager class, 159
AcApLongTransactionManager class, 70
AcApLongTransactionReactor class, 70
AcApProfileManagerReactor class, 587
AcAxOleLinkManager class, 607
AcBr library, 751
    class hierarchy, 755
    containment classes, 762
    entity classes, 761
    enumerated types, 764
    mesh classes, 762
    traverser classes, 762
accessing
    AutoLISP variables, 250
    current document and related objects, 428
    databases with noncurrent documents, 429
    symbol tables, 242
    system variables, 249
AcDb library, 12
    class hierarchy, 13
acdb15.lib library, 10
AcDb2dPolyline class, 137
    setElevation() function, 137
AcDb2dPolylineVertex class, 137
acdbAngToF() function, 267
acdbAngToS() function, 266
AcDbBlockTable class, 147
AcDbBlockTableRecord class, 99
    appendAcDbEntity() function, 85
AcDbBlockTableRecordIterator class, 152
AcDbCircle class, creating in AutoCAD, 23
AcDbCompositeFilteredBlockIterator class, 78
AcDbCurve class, 137
    overriding functions, 295
AcDbDatabase class, 394
    deep clone operation, 467, 470
    deepCloneObjects() function, 468
    dwgFileWasSavedByAutodeskSoftware() function, 80
    external reference pre- and post-processing, 75
    insert() function, 65, 468
    restoreForwardingXrefSymbols() function, 76
    restoreOriginalXrefSymbols() function, 76
    wblock() function, 63
    wblockCloneObjects() function, 70
    xrefBlockId() function, 76
AcDbDatabaseReactor class, 394, 396
    notification events, 398
AcDbDatabaseSummaryInfo class, 79
    custom fields, 79
    predefined fields, 79
AcDbDictionary class, 144
AcDbDimStyleTable class, 147
acdbDisToF() function, 267
acdbDxfOutAsR14() function, 62
acdbEntDel() function, 216, 223
acdbEntGet() function, 223, 235
acdbEntGetX() function, 235, 238, 278

AcDbEntity class, 98, 355, 363
acedSSGet() function, 111
acedSSNameX() function, 111
colorIndex() function, 102
default protocol extension class and, 370
deriving a custom entity class, 383
deriving a custom entity class from, 349
extending entity functionality, 370
using AcEdJig, 371, 383
deriving entities from, 349
draw() function, 105
explode() function, 105, 123–124
functions rarely overridden, 352
functions usually overridden, 351
getEcs() function, 136
getGeomExtents() function, 105
getGripPoints() function, 105
getGsMarkersAtSubentPath() function, 106
getOsnapPoints() function, 105–107
getStretchPoints() function, 105
getSubentPathsAtGsMarker() function, 106
getTransformedCopy() function, 105, 107
graphics generation and, 455
Graphics System Markers, 109
highlight() function, 106, 111
intersectWith() function, 105, 107–108
layer() function, 104
layerId() function, 105
linetype() function, 103
linetypeScale() function, 103
list() function, 105
moveGripPointsAt() function, 105
moveStretchPointsAt() function, 105
proxy object class derived from, 388
setColorIndex() function, 102
setLayer() function, 104
setLinetype() function, 103
setLinetypeScale() function, 103
setVisibility() function, 104
subentPtr() function, 106, 112
transformation functions, 363
transformBy() function, 105, 107
viewportDraw() function, 105
visibility() function, 104
worldDraw() function, 105
AcDbEntityReactor class, 397
acdbEntLast() function, 249
acdbEntMake() function, 149, 228–229, 232
acdbEntMod() function, 227–228, 233
acdbEntNext() function, 214, 249
acdbEntUpd() function, 234
acdbFail() function, 527
AcDbFilter class, 78
AcDbFilteredBlockIterator class, 78
AcDbFullSubentPath class, 110
acdbGetSummaryInfo() function, 80
acdbGetSummaryInfoManager() function, 80
AcDbGroup class
newIterator() function, 154
setColor() function, 154
setLayer() function, 154
setLinetype() function, 154
setVisibility() function, 154
acdbHandEnt() function, 216, 223, 240
AcDbHostApplicationServices class, 658
AcDbHyperlink class, 139
AcDbHyperlinkCollection class, 139
AcDbHyperlinkPE class, 139
AcDbIdMapping class, 508
AcDbIndex class, 78
AcDbIndexFilterManager namespace, 78
acdbInters() function, 254
AcDbIsPersistentReactor class, 402
acdbIsPersistentReactor() function, 402
AcDbLayerTable class, 147
AcDbLayerTableRecord class, 149
creating and modifying, 150
AcDbLayout class, 156, 159
AcDbLayoutManager class, 159
AcDbLayoutManagerReactor class, 159
AcDbLine class, creating in AutoCAD, 23
AcDbLinetypeTable class, 147
AcDbLongTransaction class, 69
AcDbLongTransWorkSetIterator class, 70
AcDbMatchProperties class, 516
as protocol extension class, 370
AcDbMline class, 156
AcDbMlineStyle class, 156
AcDbObject class, 82, 394–395
addReactor() function, 397
custom notification and, 398
deep cloning with, 467
deriving from, 291
erase() function, 94
new() function, 85
notification events, 398
overriding functions, 292
proxy object class derived from, 388
reactors, 395, 402
setXData() function, 86
undo mechanism and, 455
wblockClone() function, 468
xData() function, 86
AcDbObjectId class, 469
AcDbObjectIds, translating ads_names to, 83
AcDbObjectReactor class, 394, 397
notification
events, timing of, 410
functions, 397–398
transient reactors derived from, 402
acdbOpenObject() function, 82
with close() function, 454
AcDbPlotSettings class, 159
AcDbPlotSettingsValidator class, 159

AcDbProxyEntity class, 388
AcDbProxyObject class, 388, 391
acdbPutSummaryInfo() function, 80
acdbRegApp() function, 237–238
AcDbRegAppTable class, 147
acdbRToS() function, 266
acdbSaveAsR13() function, 63
acdbSaveAsR14() function, 63
AcDbSummaryInfoManager class, 80
AcDbSummaryInfoReactor class, 80
acdbTblNext() function, 242
acdbTblSearch() function, 242–243
AcDbTextStyleTable class, 147
AcDbTextStyleTableRecord class, 469
    associating with AcGiTextStyle, 707
AcDbTransactionManager class, 663
    undo mechanism and, 455
AcDbUCSTable class, 147
AcDbViewportTable class, 147
AcDbViewTable class, 147
AcDbWblockCloneFiler class, 488
acdbXdRoom() function, 239
acdbXdSize() function, 239
AcDbXrecord class, 161
AcDbXrefFileLock class, 76
acdbXrefReload() function, 76
AcEd library, 12
    class hierarchy, 12
acedAlert() function, 527
acedapi.lib library, 10
acedArxLoaded() function, 533
acedArxUnload() function, 533
acedCmd() function, 246–247
acedCommand() function, 246–247
acedDefun() function, 524, 527
acedDragGen() function, 212, 259, 263, 535
acedEntSel() function, 214, 216, 259
acedFindFile() function, 251
acedGetArgs() function, 526
acedGetDist() function, 259
acedGetFileD() function, 252
acedGetFunCode() function, 526
acedGetInput() function, 231
acedGetKword() function, 259
acedGetPoint() function, 259
acedGetReal() function, 263
acedGetString() function, 259
acedGetSym() function, 250, 526
acedGetVar() function, 249
acedGraphScr() function, 275
acedGrDraw() function, 275
acedGrRead() function, 275
acedGrText() function, 275
acedGrVecs() function, 212, 535
acedInitGet() control bits, 260
acedInitGet() function, 259–260, 263
AcEdInputContextReactor class, 569
AcEdInputPointManager class, 567
acedInvoke() function, 528
AcEditorReactor class, 75, 395–396, 398, 504
    notification functions, 398
    transactions and, 453
AcEdJig class, 371, 383
    adding entity to the database, 378
    deriving from, 350, 371
    drag loop, 372, 374
    drag() function, 372
    general steps for using, 371
    implementing sampler(), update(), and entity() functions, 375, 378
        cursor types (table), 375
        display prompt, 375
        keyword list, 375
        user input controls (list), 377
    sample code, 378, 383
    setting up parameters for drag sequence, 372
acedMenuCmd function(), 274
acedNEntSel() function, 112, 214, 217, 259
acedNEntSelP() function, 112, 214, 259, 535
acedOsnap() function, 252, 535
acedPrompt() function, 273
acedPutSym() function, 250, 526
acedRedraw() function, 275
acedRegFunc() function, 529
acedSetVar() function, 249
acedSSAdd() function, 209, 214
acedSSDel() function, 209
acedSSFree() function, 203
acedSSGet() function, 111, 200
    DXF group codes, 205
    example, 201
    selection set options, 201
acedSSLength() function, 210
acedSSMemb() function, 210
acedSSName() function, 210
    example, 210
acedSSNameX() function, 111
acedSSXform() function
    example, 211
acedTablet() function, 276
acedTextBox() function, 254
acedTextPage() function, 275
acedTextScr() function, 275
acedTrans() function, 271
acedUsrBrk() function, 264
acedVports() function, 253
acedXformSS() function, 211, 535
AcGe library, 14, 725
    class hierarchy, 15, 727
acge15.lib library, 10
AcGeFileIO class, 746
AcGeFiler class, 746
AcGeLibVersion class, 746

AcGeTol class, 729
AcGi library, 13, 683
API for ObjectDBX, 664
class hierarchy, 14, 686
acgiapi.lib library, 10
AcGiDrawable class, 684
AcGiEdgeData class, 699
AcGiFaceData class, 699
AcGiFillType, 691
AcGiSubEntityTraits class, 354–355, 691
setting attribute values, 355
AcGiTextStyle class, 704
associating with AcDbTextStyleTableRecord, 707
AcGiViewport class, 355
AcGiViewportGeometry class, 355
AcGiWorldDraw class, 354
AcGiWorldGeometry class, 354
AcGix API, 665
AcGixSimpleView class, 667
AcGixWhipView class, 668
AutoCAD features not supported, 666
getDatabaseMutex() function, 668
releaseDatabaseMutex() function, 668
representing TrueType fonts in 3D space, 666
SimpleView, 667
TrueType font elaboration, 666
using the database mutex, 668
ViewAcDb viewer, 669
WhipView, 668
acgixAllocateWhipView() function, 669
AcGixSimpleView class, 667
AcGixWhipView class, 668
acProfileManagerPtr() function, 586
acquireAngle() function, 376
acquireDist() function, 376
acquirePoint() function, 376
acquireXXX() function, 376
AcRx library, 10
class hierarchy, 11
ACRX_DECLARE_MEMBERS macro, 286
ACRX_DXF_DEFINE_MEMBERS macro, 388, 390
ACRX_NO_CONS_DEFINE_MEMBERS macro, 287
ACRX_X_CALL macro, 515
acrx15.lib library, 10
acrxAbort() function, 527
AcRxDictionary class, 11
AcRxDLinkerReactor class, 396
acrxEntryPoint() function, 36, 184–185, 770
AcRxObject class, 10, 394
cloning with, 468
function overriding, 294
acrxProductKey() function, 635
active document, 418
ActiveX Automation, 593
AcAxOleLinkManager class, 607
accessing COM interfaces from ObjectARX, 595
adding custom objects and entities to object model, 617
adding functionality to object model, 616
additional requirements for COM Objects, 609
ATL templates, 611
ATL templates provided by Autodesk, 611
AutoCAD implementation, 605
axtempl.h file, 611
building and registering a COM DLL, 621
creating a registry file, 613
creating the COM Object, 608
document locking, 612
exposing Automation functionality, 615
IAcadBaseObject interface, 606
implementation of Automation objects, 610
interacting with AutoCAD, 611
MFC access to AutoCAD ActiveX Automation, 595
non-MFC access to AutoCAD ActiveX Automation, 599
registry layout for COM objects, 609
relationship between AcDbObjects and Automation objects, 605
setting up an ATL project file, 615
writing a COM wrapper, 616
AcTransaction class, 451, 663
for newly created objects, 454
for obtaining object pointers from object IDs, 453
open and close mechanism with, 455
AcTransactionManager class, 663
flushGraphics() function, 455
for newly created objects, 454
graphics generation and, 455
nesting transactions with, 451
obtaining, 451
queueForGraphicsFlush() function, 455
AcTransactionReactor class, 396, 663
actrTransactionManager macro, 451
AcUi library
button classes, 182
CAdUiPickButton class, 182
CAdUiSelectButton class, 182
combo box controls, 178
CAcUiAngleComboBox class, 179
CAcUiNumericComboBox class, 179
CAcUiStringComboBox class, 179
CAcUiSymbolComboBox class, 179
control bar classes, 176
CAcUiDockControlBar class, 177

AcUi library (*continued*)
dialog classes, 175
CAcUiAlertDialog class, 176
CAcUiDialog class, 175
CAcUiFileDialog class, 176
CAcUiTabChildDialog class, 175
CAcUiTabMainDialog class, 175
edit controls, 177
CAcUiAngleEdit class, 177
CAcUiEdit class, 177
CAcUiHeaderCtrl class, 178
CAcUiListBox class, 178
CAcUiListCtrl class, 178
CAcUiNumericEdit class, 177
CAcUiStringEdit class, 177
CAcUiSymbolEdit class, 178
hierarchy, 173
most recently used combo boxes, 179
CAcUiArrowHeadComboBox class, 180
CAcUiColorComboBox class, 180
CAcUiLineWeightComboBox class, 180
CAcUiMRUComboBox class, 179
CAcUiMRUListBox class, 181
CAcUiPlotStyleNamesComboBox class, 181
CAcUiPlotStyleTablesComboBox class, 181
overview, 172
acui.h file, 172
acutAngle() function, 253
acutBuildList() function, 202, 247, 551
acutCvUnit() function, 269
acutDistance() function, 253
acutGetVar() function, 586
acutPolar() function, 253
acutPrintf() function, 273
acutRelRb() function, 549
acutWcMatch() function, 278–279
acutWcMatchEx() function, 279
ADC. *See* AutoCAD DesignCenter API
AddExtendedTabs() function, 183
adding
entities to the database, 378
object-specific data, 86
addPersistentReactor() function, 396, 402
AddTab() function, 183–184
addX() function, 514
adesk.h file, 771
ADS functions, 522
ads.h file, 771
ads_binary structure, 554
ads_command() function, 452
ads_matrix data type, 211
ads_name, 200
translating AcDbObjectIds to, 83
ads_point data type, 534
ads_point_set() macro, 534
ads_real data type, 534
adscodes.h file, 168, 771
adsdef.h file, 771
adsdlg.h file, 771, 792
adslib.h file, 771
AdUi library
button classes, 181
CAdUiBitmapButton class, 181
CAdUiBitmapStatic class, 182
CAdUiDropSite class, 182
CAdUiOwnerDrawButton class, 181
CAdUiToolButton class, 182
combo box controls, 178
CAdUiComboBox class, 178
control bar classes, 176
CAdUiDockControlBar class, 176
dialog classes, 174
CAdUiDialog class, 175
CAdUiFileDialog class, 175
CAdUiTabChildDialog class, 175
CAdUiTabMainDialog class, 175
edit controls, 177
CAdUiEdit class, 177
hierarchy, 173
messages, 174
overview, 172
tab extensibility, 176
CAdUiTab class, 176
CAdUiTabExtensionManager class, 176
tip windows, 174
CAdUiDrawTextTip class, 174
CAdUiTextTip class, 174
CAdUiTipWindow class, 174
adui.h file, 172
AFX_EXTENSION_MODULE, 170
AfxSetResourceHandle() function, 170
AIG. *See* Application Interoperability Guidelines
angle conversions, 266
examples, 268
angles, between 3D vectors, 731
anonymous blocks, 232
append() function, 378
appendAcDbEntity() function, 498
Application Interoperability Guidelines (AIG), 2
application names
registering, 237
registering, example, 238
applications
basic example, 39
communicating between, 528
configuration, 585
creating, 29–30
debugging with dynamic MFC, 169
execution context, 436

applications (*continued*)
external, 532
initializing, 37
interoperability, 2
listing loaded, 43, 53
loading, 43, 53, 772
MDI requirements, 423–424
messages sent to ObjectARX, 31
proxy objects and, 388, 391
reactors in, 394
result codes, 529
running from AutoLISP, 55
sequence of events in, 35
unloading, 38, 44, 53, 391, 515
unlocking, 44
using protocol extension functionality, 515
Windows system registry entries in, 46
applyPartialUndo() function, 325–326
arc primitive, 703
argument lists in AutoLISP and C, 522
ARX command, 53
AsdkPoly class
overriding intersectWith() function, 365
transformation functions and, 363
with grip points, 360
assertNotifyEnabled() function, 297
assertReadEnabled() function, 297, 300, 302
assertWriteEnabled() function, 297, 300, 302, 324, 327, 411
assoc AutoLISP function, 227
associating
AcDbTextStyleTableRecord with AcGiTextStyle, 707
hyperlinks with entities, 139
association list, 226
asynchronous interaction from an automation client, 611
ATL
setting up an ATL project file, 615
templates, 611
AttachInstance() function, 170
attribute values, 355
auditing extended data handles, 240
AutoCAD
AutoCAD DesignCenter API, 634
classes, protocol extension and, 511
cloning phase, 477
commands
boundaries in transactions, 452
using deep clone and wblock clone, 476
using wblock clone, 476
creating circle entities in, 23
creating entry points for, 36
creating group entities in, 24
creating instances of entities, 125
creating layer entities in, 24
AutoCAD (*continued*)
creating line entities in, 23
creating objects in, 22
creating registry subkeys and values, 48
database overview, 20
demand loading features, for proxy objects, 388
ERRNO system variable, 527
features that use COM, 594
key components, 20
proxy entity messages in, 389
Release 12 entities, 100
responding to messages from, 31
translation phase, 477
undo in transactions, 455
AutoCAD command functions, 246
AutoCAD DesignCenter API, 634
applications registry branch example, 636
applications registry key, 635
CLASSID registration, 637
custom content example, 640
extensions registry branch example, 637
extensions registry key, 636
IAcDcContentBrowser interface, 634
IAcDcContentBrowser interface functions, 638
IAcDcContentFinder interface, 635
IAcDcContentFinderSite interface, 634
IAcDcContentView interface, 634
IAcDcContentView interface functions, 639
IAcPostDrop interface, 635
implementing interfaces, 638
providing custom content, 640
registry requirements, 635
using the acrxProductKey() function, 635
AutoCAD Release 12
saving to DWG file, 356
AutoLISP
comparison of AutoLISP calls to ObjectARX calls, 522
defining functions, 524
expressions in user input, 259
functions
(assoc), 227
(command), 246
(entget), 238
(vports), 253
returning values to, 265
linked lists, 551
running applications from, 55
variables
accessing from ObjectARX, 250
setting to nil, 251
automatic undo, 325
automation. *See* ActiveX Automation
axtempl.h file, 611, 617

B

beginClose() function, 398
beginDeepClone() function, 489, 504–505, 507
beginDeepCloneXlation() function, 491, 504–505, 507
    ID map (table), 492, 504
    notification functions, 507
beginDxfIn() function, 398
beginInsert() function, 507
beginSave() function, 398
beginWblock() function, 490, 505
BHATCH command, 370
blanking out, 275
BLOCK command, 149
    cloning application, 476
block references with attributes, creating, 129
block table, 147, 149
block table records
    appending function to, 378
    creating, 126
    creating, with attribute definitions, 126
    iterating through, 133
blocks, anonymous, 232
boundaries of transactions, 452
boundary representation library. *See* AcBr library
bounding box for text, 254
building an ownership hierarchy, 312
button classes
    AcUi, 182
    AdUi, 181

C

CAcExtensionModule class, 170
    DLL initialization and termination, 170
    example, 171
    resource tracking, 170
CAcModuleResourceOverride class, 171
    example, 171
CAcUiAlertDialog class, 176
CAcUiAngleComboBox class, 179
CAcUiAngleEdit class, 177
CAcUiArrowHeadComboBox class, 180
CAcUiColorComboBox class, 180
CAcUiDialog class, 175
CAcUiDockControlBar class, 177
CAcUiEdit class, 177
CAcUiFileDialog class, 176
CAcUiLineWeightComboBox class, 180
CAcUiMRUComboBox class, 179
CAcUiMRUListBox class, 181
CAcUiNumericComboBox class, 179
CAcUiNumericEdit class, 177
CAcUiPickButton class, 182
CAcUiPlotStyleNamesComboBox class, 181
CAcUiPlotStyleTablesComboBox class, 181
CAcUiSelectButton class, 182
CAcUiStringComboBox class, 179
CAcUiStringEdit class, 177
CAcUiSymbolComboBox class, 179
CAcUiSymbolEdit class, 178
CAcUiTabChildDialog class, 175
CAcUiTabMainDialog class, 175
CAdUiBitmapButton class, 181
CAdUiBitmapStatic class, 182
CAdUiComboBox class, 178
CAdUiDialog class, 175
CAdUiDockControlBar class, 176
CAdUiDrawTipText class, 174
CAdUiDropSite class, 182
CAdUiEdit class, 177
CAdUiFileDialog class, 175
CAdUiHeaderCtrl class, 178
CAdUiListBox class, 178
CAdUiListCtrl class, 178
CAdUiOwnerDrawButton class, 181
CAdUiTab class, 176
CAdUiTabChildDialog class, 175
CAdUiTabExtensionManager class, 176
CAdUiTabMainDialog class, 175
CAdUiTextTip class, 174
CAdUiTipWindow class, 174
CAdUiToolButton class, 182
calibrating tablets, 276
callback functions supporting DCL, 785
callback reason values in DCL, 788
cancel() function, 328, 411, 455
cancelled() function, 397, 410–411
CECOLOR system variable, 66
CELTSCALE system variable, 66, 103
CELTYPE system variable, 102
changing callback modes and values for DCL tiles, 796
character conversion and testing functions, 270
checkIn() function, 321
checkOut() function, 321–322
circularArc() function, 703
class descriptor objects, 285
class hierarchy
    AcBr, 755
    AcDb, 13
    AcEd, 12
    AcGe, 15, 727
    AcGi, 14, 686
    AcRx, 11
    AcUi, 173
    AdUi, 173
classes
    AcApDocManager, 417
    AcApDocManagerReactor, 418
    AcApDocument, 61
    AcApDocumentIterator, 417
    AcApLayoutManager, 159
    AcApLongTransactionManager, 70

classes (*continued*)
AcApLongTransactionReactor, 70
AcApProfileManager, 586
AcApProfileManagerReactor, 587
AcBr containment, 762
AcBr entity, 761
AcBr mesh, 762
AcBr traverser, 762
AcDb2dPolyline, 137, 285
AcDb2dPolylineVertex, 137, 285
AcDb3dPolyline, 285
AcDb3dPolylineVertex, 285
AcDbArc, 284
AcDbAttribute, 284
AcDbAttributeDefinition, 284
AcDbBlockBegin, 284
AcDbBlockEnd, 284
AcDbBlockReference, 284
AcDbBlockTable, 147
AcDbBlockTableRecordIterator, 152
AcDbCircle, 284
AcDbCompositeFilteredBlockIterator, 78
AcDbCurve, 137, 284
AcDbDatabaseReactor, 284
AcDbDatabaseSummaryInfo, 79
AcDbDimStyleTable, 147
AcDbEntity, 98, 284, 363
AcDbEntityReactor, 284
AcDbFace, 284
AcDbFaceRecord, 285
AcDbFilter, 78
AcDbFilteredBlockIterator, 78
AcDbFullSubentPath, 110
AcDbGroup, 284
AcDbHostApplicationServices, 658
AcDbHyperlink, 139
AcDbHyperlinkCollection, 139
AcDbIndex, 78
AcDbLayerTable, 147
AcDbLayerTableRecord, 149
AcDbLayout, 156, 159
AcDbLayoutManager, 159
AcDbLayoutManagerReactor, 159
AcDbLine, 284
AcDbLinetypeTable, 147
AcDbLongTransaction, 69
AcDbLongTransWorkSetIterator, 70
AcDbMatchProperties, 516
AcDbMInsertBlock, 284
AcDbMline, 156
AcDbMlineStyle, 156
AcDbObject, 284
AcDbObjectId, 469
AcDbObjectReactor, 284
AcDbPlotSettings, 159
AcDbPlotSettingsValidator, 159

classes (*continued*)
AcDbPoint, 284
AcDbPolyFaceMesh, 285
AcDbPolyFaceMeshVertex, 285
AcDbPolygonMesh, 285
AcDbPolygonMeshVertex, 285
AcDbRegAppTable, 147
AcDbSequenceEnd, 285
AcDbShape, 284
AcDbSolid, 284
AcDbSummaryInfoManager, 80
AcDbSummaryInfoReactor, 80
AcDbSymbolTable, 284
AcDbSymbolTableRecord, 284
AcDbText, 284
AcDbTextStyleTable, 147
AcDbTrace, 284
AcDbTransactionManager, 663
AcDbUCSTable, 147
AcDbViewport, 284
AcDbViewportTable, 147
AcDbViewTable, 147
AcDbXrecord, 161
AcDbXrefFileLock, 76
AcDbXxxDimension, 284
AcEdInputContextReactor, 569
AcEdInputPointManager, 567
AcEditorReactor, 284, 398, 504
AcEditorReactor class, 75
AcEdJig, 284, 371
AcGeFileIO, 746
AcGeFiler, 746
AcGeLibVersion, 746
AcGeTol, 729
AcGiDrawable, 684
AcGiEdgeData, 699
AcGiFaceData, 699
AcGiSubEntityTraits, 691
AcGiTextStyle, 704
AcRxDictionary, 11
AcRxObject, 10, 284
AcRxService, 284
AcTransaction, 663
AcTransactionManager, 663
AcTransactionReactor, 284, 663
CAcExtensionModule, 170
CAcModuleResourceOverride, 171
CAcUiAlertDialog, 176
CAcUiAngleComboBox, 179
CAcUiAngleEdit, 177
CAcUiArrowHeadComboBox, 180
CAcUiColorComboBox, 180
CAcUiDialog, 175
CAcUiDockControlBar, 177
CAcUiEdit, 177
CAcUiFileDialog, 176

classes (*continued*)
CAcUiLineWeightComboBox, 180
CAcUiMRUComboBox, 179
CAcUiMRUListBox, 181
CAcUiNumericComboBox, 179
CAcUiNumericEdit, 177
CAcUiPickButton, 182
CAcUiPlotStyleNamesComboBox, 181
CAcUiPlotStyleTablesComboBox, 181
CAcUiSelectButton, 182
CAcUiStringComboBox, 179
CAcUiStringEdit, 177
CAcUiSymbolComboBox, 179
CAcUiSymbolEdit, 178
CAcUiTabChildDialog, 175
CAcUiTabMainDialog, 175
CAdUiBitmapButton, 181
CAdUiBitmapStatic, 182
CAdUiComboBox, 178
CAdUiDialog, 175
CAdUiDockControlBar, 176
CAdUiDrawTipText, 174
CAdUiDropSite, 182
CAdUiEdit, 177
CAdUiFileDialog, 175
CAdUiHeaderCtrl, 178
CAdUiListBox, 178
CAdUiListCtrl, 178
CAdUiOwnerDrawButton, 181
CAdUiTab, 176
CAdUiTabChildDialog, 175
CAdUiTabExtensionManager, 176
CAdUiTabMainDialog, 175
CAdUiTextTip, 174
CAdUiTipWindow, 174
CAdUiToolButton, 182
containers hierarchy, 145
control bar, 176
CPropertyPage, 175
CPropertySheet, 175
creating custom, 31
creating MFC dialog, 189
custom class derivation, 284
custom class function declaration, 286
declaring and defining protocol extensions, 512
document management, 417
evaluation, 737
implementing deepClone() for custom classes, 476
initialization function, 289
line, 732
matrix, 730
plane, 732
point, 730
reactors, 394

classes (*continued*)
renaming, 346
runtime class identification, 285
vector, 730
version support, 343
clear_reactors() function, 399
clip boundaries in AcGi, 723
clone() function, 468
versus deepClone() function, 468
cloning
deep, 467
versus deep cloning, 468
deep clone types and duplicate record cloning, 71
editor reactor notification functions, 504
exploding and, 477
ID map, 470
inserting, 504
key concepts, 469
objects from different owners, 472
overriding the deepClone() function, 484
overriding the wblockClone() function, 488
in transformation functions, 363
translating, 470
using appendAcDbEntity(), 498
close() function, 328, 411–412
for newly created objects, 454
object pointers and, 454
open and close mechanism with, 455
closing ObjectARX objects, 27
CMDACT system variable, 569–570
collections of hyperlinks, 139
colors
entity, 101
values, 66
COM
accessing COM interfaces from ObjectARX, 595
AutoCAD features that use COM, 594
building and registering a COM DLL, 621
See also ActiveX Automation
combo box controls, 178
command AutoLISP function, 246
command line prompt, 375
command stack, 40
commandEnded() function, 453, 505
commands
ARX, 53
AutoCAD, for cloning applications, 476
BLOCK, 149
boundaries (AutoCAD) in transactions, 452
displaying names in command groups, 53
EXPLODE, 124
global versus local names, 42
group names, 40
INSERT, 149

commands (*continued*)
LAYER, 149
lookup order, 42
modal, 42
multi-document, 432
nonreentrant, 431
not used in transactions, 453
registering new, 40
transparent versus modal, 42
commandWillStart() function, 453
commit-time
guidelines, 454
notification, 410
common characteristics of global functions, 522
common entity properties, 101
communicating between applications, 528
comparison
ObjectARX calls and AutoLISP calls, 522
symbol tables and dictionaries, 144
compatibility levels for SDI and MDI, 423
complex entities, 98, 134, 217, 229
component codes for entities, 226
computing points on parametric curves, 738
conditional filtering, 208
consistency checks, 76
constructing tab dialogs, 183
container objects, 143
context data, 221
context events, input, 569
context help, 174
control bar classes, 176
controlling graphics and text screens, 275
controlling long transactions, 70
controlling the display, 273
controls
creating, 189
user input, 377
conversions
angles, 266
coordinate systems, 271
numbers, 266
strings, 266
units, 269
coordinate systems
accessing, 136
conversions, 271
descriptions, 271
specifying, 271
transformation example, 273
transformations, 271
copied() function, 397, 411
COPY command, 476
copying
arrays of entities to create databases, 64
from entity to entity, 370
named blocks to create databases, 64
CPROFILE system variable, 586
CPropertyPage class, 175
CPropertySheet class, 175
createObjs() function, 313
createXrecord() function, 163
creating
AcDbLayerTableRecord example, 150
application registry keys and values, 48
applications, 29–30
block table records, 126
with attribute definitions, 126
circle entities in AutoCAD, 23
class descriptor objects, 285
classes and controls, 189
complex entities, 134
custom classes, 31
custom entities, 350
databases from existing databases, 63
databases with entities, 64
dialog handlers, 190
dictionaries, 157
entities, 25
entry points for AutoCAD, 36
instances of AutoCAD entities, 125
layer entities in AutoCAD, 24
layer groups in AutoCAD, 24
layers, 26
line entities in AutoCAD, 23
MFC dialog using App Studio, 188
objects in AutoCAD, 22
objects in ObjectARX, 25
objects of protocol extension classes, 514
ownership connection, 311
proxies, 389
registry file (*.reg*) for custom object COM server, 613
selection sets, 200
simple entities, 125
skeletons, 186
Windows system registry subkeys and values, 48
creating a database, 60
ctype.h file, 270
current database, accessing, 21
current document
accessing, 428
setting, 428
current save format, 62
cursor
specified types with AcEdJig, 371
types (table), 375
curves
AcDbCurve class, 137
AcGe functions, 733
computing points on, 738
functions, 137
tessellating, 709

custom classes
creating, 31
overriding AcDbCurve functions, 292
overriding AcDbObject functions, 292
overriding AcRxObject functions, 292
custom entities
adding, using AcEdJig, 371
deriving, 350
deriving from AcDbEntity, 349, 355, 357, 370
extending entity functionality, 370
intersecting with other entities, 369
stretch points, 361
transformation functions, 363
custom notifications, 398
custom object snap modes, 558
custom objects
LongTransaction issues, 321
proxy objects and, 388–389, 391

D

data
application-specific DCL data, 806
class version numbers, 347
class version support, 344
dynamically allocated, 546
instances per document, 416
ObjectARX-exclusive data type, 778
per-document data, 424
data types
acad_proxy_entity, 389
acad_proxy_object, 389
ads_matrix, 211
ads_name, 200
runtime identification mechanism, 283
database operations, 59
databases
accessing with noncurrent documents, 429
adding entities to, 378
closing objects, 82
color values, 66
components of, 20
creating, 60
creating by copying arrays of entities, 64
creating by copying named blocks, 64
creating databases with entities, 64
creating empty, 22
creating from existing databases, 63
current, 21
dependencies, finding, 402
document-independent databases, 439
essential objects, 22
example of operations, 67
initial values, 60
inserting, 65
layer values, 67
linetype scale values, 66
databases (*continued*)
linetype values, 66
merging, 65
multiple, 21
notification events, 398
objects overview, 82
opening objects, 82
overview, 20
ownership of objects, 85
ownership structure for entities, 99
populating, 60
primer, 19
reactors, using, 394
resolving conflicts while inserting, 65
resolving conflicts while merging, 65
restoring in external references, 76
saving, 61
setting color values, 66
setting current values, 66
setting linetype scale values, 66
setting linetype values, 66
startup values, 60
wblock cloning, 488
dbapserv.h. file, 659
DCL. *See* Dialog Control Language
DCS. *See* display coordinate system
debugging applications with dynamic MFC, 169
deep cloning, 467
calls, 505
cloning objects from different owners, 472
filing, 469
functions, 467
implementing
AutoCAD commands for, 476
cloning phase, 477, 484
filers for, 484
insert operation, 486
translation phase, 477, 484
in long transactions, 70
key concepts, 469
ownership, 468–469
typical operation, 470
using clone() versus deepClone(), 468
wblock functions and, 467–468
deepClone() function, 321, 468, 476, 484, 499
AutoCAD commands using, 476
versus clone() function, 468
cloning phase, 477
overriding, 484
translation phase, 477
deepCloneContext() function, 508
deepCloneObjects() function, 467, 470
implementing, 476
default actions for DCL dialog boxes, 786
default file format
considerations, 62
settings, 61

definition data, retrieving, 227
degeneracy of entities, 736
deleteAcRxClass() function, 391
deleting
extended data, 236
objects, 85
demand loading, 45
for ObjectDBX, 671
on AutoCAD start-up, 52
on command, 51
on detection of custom objects, 50
DEMANDLOAD system variable, 49
deriving
custom classes, 284
custom entities, 350
new classes from AcEdJig, 371
DesignCenter. *See* AutoCAD DesignCenter API
deviation() function, 709
dialog boxes
data persistency, 182
designing with DCL, 780
naming conventions, 183
dialog classes
AcUi, 175
AdUi, 174
Dialog Control Language (DCL), 780
ads_start_dialog() status code values, 793
application-specific data, 806
callback function definitions, 793
callback functions, 785
callback reason values, 788, 794
changing callback modes and values for tiles, 796
data handling, 806
declarations, 792
default actions, 786
definitions, 792
dialog box color numbers, 801
edit box handling, 806
examples
basic dialog box, 781
hiding a dialog box, 790
hiding multiple dialog boxes, 790
image creation, 802
multiple selection lists, 799
functions not allowed while a dialog box is active, 783
handles for dialog boxes and tiles, 792
hiding dialog boxes, 789
image button handling, 803
image creation, 801
initializing modes and values for tiles, 795
list boxes, 797
list function code values, 794
list value handling, 799
nesting dialog boxes, 789
Dialog Control Language (DCL) (*continued*)
passing arguments in callback functions, 786
pop-up lists, 797
radio cluster handling, 803
setting up list boxes and pop-up lists, 797
slider handling, 804
supporting DCL dialog boxes in an application, 780
tile handling, 795
tile mode values, 795–796
dictionaries, 144, 153
adding and deleting entries, 153
comparison with symbol tables, 144
creating, example, 157
extension, 89
group, 28, 153
layout, 156
Mline style, 156
named object, 480
names, 147
setAt() function, 153
dimstyle table, 147
directory tree, ObjectARX, 16
disableSystemCursorGraphics() function, 568
display control, 273
display coordinate system (DCS), 712
definition, 272
display prompt
for drag sequence with AcEdJig, 372
setting, 375
displaying
command names, 53
entity changes, 233
proxy entities, 390
displaying menus
example, 274
functions, 274
displaying text
functions, 273
size limits, 273
DLLs
initializing, 170
terminating, 170
docking system, 176
document management classes, 417
documentation
online, 2
printed, 2
document-independent databases, 439
documents
accessing current, 428
disabling switching, 435
events, 430
execution context, 416
locking, 417

documents (*continued*)
locking for Automation requests, 612
object, 417
downgradeOpen() function, 86
drag sequences
controlling, with AcEdJig, 371
drag loop, 372
drag loop, illustrated, 374
implementing, with AcEdJig, 371
limitations on, 377
setting display prompt, 372
drag() function, 372
dragging, 378
dragging selection sets, 263
draw() function, 105, 455
drawables, 708
drawing ancillary data, 79
drawings
save functions, 63
summary information, 79
transactions and, 455
DrawTips, 174
dummy entities, 241
DWG files, 298
from earlier releases, 680
dwgIn() function, 299, 469
dwgInFields() function, 300, 325, 347
dwgOut() function, 299, 469
dwgOutFields() function, 300, 324, 327
DXF files, 298
DXF group codes, 542
extended data, 236
handle value, 240
in acedSSGet(), 205
ranges, 303
sentinel codes, 226, 235
xrecords, 162, 241
dxfIn() function, 299
dxfInComplete() function, 398
dxfInFields() function, 304, 347
dxfOut() function, 299
dxfOutFields() function, 302
dynamic properties and Object Property Manager, 630
dynamically allocated data, 546

E

ECS. *See* Eye Coordinate System
edit controls, 177
editing proxy entities, 390
editor reactor, 398
notification functions, 504
eInProcessOfCommitting message, 411, 454
ellipse, creating (example), 378, 383
empty databases, 22
enable GraphicsFlush() function, 455
enableSystemCursorGraphics() function, 568
endCalledOnOutermostTransaction() function, 454, 456
endDeepClone() function, 480, 504–505
endInsert() function, 507
endWblock() function, 505
entget AutoLISP function, 238
entities
AcDbBlockTableRecord, 99
adding to the database, 378
assigning extended data, 241
associating hyperlinks, 139
AutoCAD Release 12, 100
colors, 101
complex, 98, 134, 217, 229
component codes, 226
context, 217
context data, 221
coordinate transformation data, 217
creating, 25
creating instances of AutoCAD entities, 125
data functions, 223
defined, 98
definition data, 223
degeneracy, 736
displaying changes, 233
dummies, 241
exploding, 123
extending entity functionality, 370
extending functionality, 370
forced entity picking, 568
intersecting with other entities, 369–370
layer, 104
linetype, 102
linetype scale, 103
linetype scale in paper space, 104
linked result buffers, 221
name volatility, 200
names, 200, 214
native, 369
nested entities, 221
nested in block references, 217
not derivable, 100
object snap point functions, 357
ownership, 99, 242
properties, common, 101
proxies, 387
purging, 223
recovering, 216
regenerating, 104, 234
restoring color, 227
restoring linetype, 227
saving, 356
setting entity traits, 690
snap points, 106
stretch points, 361

entities (*continued*)
subentity traits, 691
transformation example, 222
transformation functions, 363
translation example, 218
using AcEdJig, 371, 383
adding the entity to the database, 378
deriving a new class from AcEdJig, 371
drag loop, 372
general steps for using AcEdJig, 371
implementing sampler(), update(), and entity() functions, 375
sample code, 378, 383
setting up parameters for drag sequence, 372
using handles, 216
visibility, 104
entity coordinate system
definition, 272
entity() function, 373
implementing, 375, 378
overriding, 371
returning points to entity with, 378
entry points, creating for AutoCAD, 36
enumerated types, AcBr, 764
erase() function, 94, 328
erased() function, 397–398, 411
erasing objects, 94
error codes
eWasErased, 94
when opening objects, 84
error handling, 55
AutoCAD system variable ERRNO, 527
filing objects to DXF and DWG files, 300
in selection sets, 214
invoked functions, 532
ObjectARX, 527
RTERROR, 205
essential database objects, 22
evaluating external functions, 526
evaluation classes, 737
events
document, 430
immediate events, 410
input context events, 569
notification, 394, 398
sequence of, in applications, 35
time-commit events, 410
eWasErased error code, 94
eWasNotifying message, 453
eWasNotOpenForWrite message, 411
examples
accessing AutoLISP variables, 250
AcDbLayerTableRecord creating and modifying, 150
acedCmd(), 248
acedCommand(), 247
examples (*continued*)
acedFindFile(), 251
acedGetFileD(), 252
acedGetInput(), 263
acedGetSym(), 250
acedInitGet(), 261
acedOsnap, 252
acedPutSym(), 250
acedSSGet(), 201
acedSSName() usage, 210
acedSSXform() usage, 211
acedTextBox(), 256
AcGe persistency, 746
acutBuildList(), 248
adding code to handlers, 191
adding members to selection sets, 214
angle conversions, 268
anonymous blocks, 232
application name registration, 238
AutoCAD DesignCenter API applications registry branch, 636
AutoCAD DesignCenter API custom content, 640
AutoCAD DesignCenter API extensions registry branch, 637
basic application, 39
building object dependencies, 404
CAcExtensionModule class, 171
CAcModuleResourceOverride class, 171
calling external functions, 528
checking entities in and out of a database, 71
class data and xdata version support, 347
class version support, 344, 346
clip boundary, 724
cloning and translation, 477
cloning objects from different owners, 472
COM ActiveX Automation access using MFC, 595
COM ActiveX Automation access without MFC, 599
constructing an extensible custom tab dialog, 184
coordinate system transformation, 273
creating a mesh, 697
creating an application skeleton, 186
creating an MFC dialog using App Studio, 188
creating block table records, 127
creating classes and controls, 189
creating complex entities, 134, 229
creating custom entities, 378
creating dialog handlers, 190
creating dictionaries, 157
creating simple AutoCAD entities, 125
curve functions, 138
custom object classes, 338

examples (*continued*)
custom object header file, 338
custom object snap mode, 561
custom object source file, 338
database operations, 67
database reactors, 399
DCL basic dialog box, 781
DCL data handling, 806
DCL hiding a dialog box program, 790
DCL hiding multiple dialog boxes, 790
DCL image creation, 802
DCL multiple selection lists, 799
deep clone operation, 470
deepClone() function, 499
definition data retrieval, 227
definition data retrieving and printing, 224
deleting extended data, 236
displaying entity changes, 234
displaying menus, 274
dragging selected objects, 263
dwgInFields() function, 301
dwgOutFields() function, 300
DXF group code handling, 543
dxfInFields() function with order dependence, 307
dxfInFields() with order independence, 305
dxfOutFields() function, 304
entity handles, 216
entity names, 210
entity transformation, 222
entity translation, 218
extending built-in tab dialogs, 185
extension dictionary global functions, 91
file dialog box, 252
file search, 251
finding persistent reactors, 402
finding viewports, 244
geometric functions, 254
gripping and transforming objects, 359
group and group dictionary functions, 154
handling hard references to AcDbEntities during wblockClone(), 502–503
highlighting a subentity, 112
highlighting nested block references, 116
hyperlinks, 140
input context events, 571
input options, 261
input point filter and monitor, 577
inserting blocks with references into drawings, 129
interacting with AutoCAD using ActiveX Automation, 612
intersecting functions, 365
iterating over dictionary entries, 158
iterating through block table records, 133
iterating through vertices in a polyline, 135
keywords comparison, 263

examples (*continued*)
linked lists for AutoLISP, 551
linking result buffers, 546
long transactions, 71
LongTransaction, 322
matrix composition, 213
matrix multiplication, 220
MDI-Aware example application, 440
message map, 168
migrating ADS Programs to ObjectARX, 773
named object dictionary, 481
nested transactions, 457
numeric conversions, 268
object snap, 252
overriding resources, 171
overriding subsidiary functions, 328
overriding the deepClone() function, 484
overriding the wblockClone() function, 488
ownership hierarchy, 313
pausing for input in commands, 248
pick points in commands, 249
profile manager, 588
protocol extension, 516
protocol extension class for custom object snaps, 559
reactor adding behavior to a WBLOCK command, 507
reading a drawing, 67
registry file (*.reg*) for custom object COM server, 613
resbuf, 204–205
retrieving blocks, 242
retrieving extended data for specific applications, 238
returning a value to AutoLISP, 266
reversing damage to the graphics screen, 276
sample application using the AcBr library, 766
saving a drawing, 67
scaling selections, 211
searching for files, 251
searching resbufs for specific DXF codes, 227
selecting using conditional tests, 208
selecting using extended data, 206
selecting using relational tests, 207
set an AutoLISP variable to nil, 251
setting AutoLISP variables, 250
setting entity names, 539
setting the Model Space viewport for ObjectDBX, 678
setting up an ATL project file for a COM wrapper, 615
shell with color and visibility, 701
simple block table records, 126
snap mode, 357

examples (*continued*)
string conversions, 267
system variable access, 249
tablet calibration, 276
text coordinates, 256
text with bounding box, 705
transformation, 712
transformation functions, 363
transforming text boxes, 257
unit conversion, 269
user input, 261
user input break or cancel, 264
using a transient editor reactor, 508
using AcGi, 693
using appendAcDbEntity() during cloning, 498
using the AcBr library, 766
viewport descriptors, 253
wblockClone() function, 500
wild-card comparisons, 278
xrecord functions, 163
execution contexts
application, 436
document, 416
EXPLODE command, 125, 370
cloning application, 477
explode() function, 105, 123–124, 369–370
exploding entities, 123, 370
extended data
application names, 237
assigning to entities, 241
auditing handles, 240
deleting, 236
DXF group codes, 236
filtering for, 206
handle translation, 240
limit in size, 239
managing memory use, 239
notes, 235
organization, 235
protecting application data, 239
retrieving, 238
sentinel code, 235
using handles, 240
versus xrecords, 242
See also xdata
extended entity data (EED or xdata). *See* xdata
extending entity functionality, 370
extension dictionaries, overview, 89
external functions, 524
external references
block table records, 76
clipped, 77
consistency checks, 76
contents, 74
file locks, 76
external references (*continued*)
overview, 74
pre- and post-processing, 75
reloading, 76
restoring databases, 76
restoring resolved symbols, 76
separate databases, 74
Eye Coordinate System (ECS), 712

F

F1 help, 174
figures
cloning ownership connections, 469
coordinate transformations, 711
creating circle entities in AutoCAD, 23
creating group entities in AutoCAD, 24
creating layer entities in AutoCAD, 24
creating line entities in AutoCAD, 23
input point filtering and monitoring, 575
key components of AutoCAD databases, 20
mesh edge ordering, 696
mesh face properties, 698
ownership references, 307
points returned by acedTextBox(), 255
regenerating a drawing, 685
sequence of events in application, 35
file locks
external references, 76
file searching, 251
filers, for deep cloning, 484, 488
files
adsdlg.h, 792
DWG, 298, 389
DXF, 298, 389
formats for saving drawings, 61
locking, 76
saving objects to files, 298
setting the default formats, 61
filing objects, 95
filing, and cloning, 469
fill type, primitive, 691
filtering
conditional, 208
extended data, 206
multiple properties, 204
relational tests, 207
selection sets, 203
wild-card patterns, 205
filters
chaining input point filters, 576
input point, 575
finding viewports example, 244
flushGraphics() function, 455
forced entity picking, 568
funcload() function, 529

functions
abortDeepClone(), 504
abortInsert(), 507
abortWblock(), 505
accessing AutoLISP variables, 250
acdbAngToF(), 267
acdbAngToS(), 266
acdbDisToF(), 267
acdbDxfOutAsR14(), 62
acdbEntDel(), 216, 223
acdbEntGet(), 223, 235
acdbEntGetX(), 235, 238, 278
acdbEntLast(), 249
acdbEntMake(), 149, 228–229, 232
acdbEntMod(), 227–228, 233
acdbEntNext(), 214, 249
acdbEntUpd(), 234
acdbFail(), 527
acdbGetSummaryInfo(), 80
acdbGetSummaryInfoManager(), 80
acdbHandEnt(), 216, 223, 240
acdbInters(), 254
acdbIsPersistentReactor(), 402
acdbOpenObject(), 82
acdbPutSummaryInfo(), 80
acdbRegApp(), 237–238
acdbRToS(), 266
acdbSaveAsR13(), 63
acdbSaveAsR14(), 63
acdbTblNext(), 242
acdbTblSearch(), 242–243
acdbXdRoom(), 239
acdbXdSize(), 239
acdbXrefReload(), 76
acedAlert(), 527
acedArxLoaded(), 533
acedArxUnload(), 533
acedCmd(), 246–247
acedCommand(), 246
acedDefun(), 524, 527
acedDragGen(), 212, 259, 263, 535
acedEntSel(), 214, 216, 259
acedFindFile(), 251
acedGetArgs(), 526
acedGetDist(), 259
acedGetFileD(), 252
acedGetFunCode(), 526
acedGetInput(), 231
acedGetKword(), 259
acedGetPoint(), 259
acedGetReal(), 263
acedGetString(), 259
acedGetSym(), 250, 526
acedGetVar(), 249
acedGraphScr(), 275
acedGrDraw(), 275

functions (*continued*)
acedGrRead(), 275
acedGrText(), 275
acedGrVecs(), 212, 535
acedInitGet(), 259–260, 263
acedInvoke(), 528
acedMenuCmd(), 274
acedNEntSel(), 112, 214, 217, 259
acedNEntSelP(), 112, 214, 259, 535
acedOsnap(), 252, 535
acedPrompt(), 273
acedPutSym(), 250, 526
acedRedraw(), 275
acedRegFunc(), 529
acedSetVar(), 249
acedSSAdd(), 214
acedSSAddl(), 209
acedSSDel(), 209
acedSSFree(), 203
acedSSGet(), 111, 200
acedSSLength(), 210
acedSSMemb(), 210
acedSSName(), 210
acedSSNameX(), 111
acedTablet(), 276
acedTextBox(), 254
acedTextPage(), 275
acedTextScr(), 275
acedTrans(), 271
acedUsrBrk(), 264
acedVports(), 253
acedXformSS(), 211, 535
acgixAllocateWhipView(), 669
acProfileManagerPtr(), 586
acquirePoint(), 376
acrxAbort(), 527
acrxEntryPoint(), 36, 184–185, 770
acrxProductKey(), 635
acutAngle(), 253
acutBuildList(), 202, 247, 551
acutCvUnit(), 269
acutDistance(), 253
acutGetVar(), 586
acutPolar(), 253
acutPrintf(), 273
acutRelRb(), 549
acutWcMatch(), 278–279
acutWcMatchEx(), 279
AddExtendedTabs(), 183
AddTab(), 183–184
addX(), 514
AfxGetResourceHandle(), 170
angle conversion, 266
appendAcDbEntity(), 85, 498
applyPartialUndo(), 325–326
assertNotifyEnabled(), 297

functions (*continued*)
assertReadEnabled(), 297, 300, 302
assertWriteEnabled(), 297, 300, 302, 324, 327
AttachInstance(), 170
AutoCAD commands, 246
AutoCAD services, 246
beginClose(), 398
beginDeepClone(), 504
beginDeepCloneXlation(), 504
beginDxfIn(), 398
beginInsert(), 507
beginSave(), 398
beginWblock(), 505
cancel(), 328
cast(), 285
character conversion and testing, 270
checkIn(), 321
checkOut(), 321–322
circularArc(), 703
clone(), 468
close(), 328
colorIndex(), 102
common characteristics of library functions, 522
conversion utilities, 266
coordinate system conversion, 271
createObjs(), 313
createXrecord(), 163
deepClone(), 321, 468, 476, 484, 499
deepCloneObjects(), 468, 470
deleteAcRxClass(), 391
desc(), 285
deviation(), 709
disableSystemCursorGraphics(), 568
displaying menus, 274
displaying text, 273
downgradeOpen(), 86
drag(), 372
draw(), 105
dwgFileWasSavedByAutodeskSoftware(), 80
dwgIn(), 299, 469
dwgInFields(), 300, 325, 347
dwgOut(), 299
dwgOutFields(), 300, 324, 327
dxfIn(), 299
dxfInComplete(), 398
dxfInFields(), 304, 347
dxfOut(), 299
dxfOutFields(), 302
enableSystemCursorGraphics(), 568
endCalledOnOutermostTransaction(), 456
endDeepClone(), 504
endInsert(), 507
endWblock(), 505
entity(), 375

functions (*continued*)
erase(), 94, 328
executing AutoCAD commands, 246
explode(), 105, 123–124, 370
external, 524
file search, 251
flushGraphics(), 455
for drawing primitives, 354
formatForSave(), 62
funcload(), 529
geometric utilities, 253
getAt(), 147
getDatabaseMutex(), 668
GetDialogName(), 184
getEcs(), 136
getFilerStatus(), 300
getGeomExtents(), 105
getGripPoints(), 105, 359
getGsMarkersAtSubentPath(), 106
GetObjectId(), 607
getOsnapInfo(), 559
getOsnapPoints(), 105–107, 357
getStretchPoints(), 105, 361
getSubentPathsAtGsMarker(), 106, 111
getTransformedCopy(), 105, 107, 363
highlight(), 106, 111
insert(), 65, 468
intersectWith(), 105, 107–108, 364, 369
isA(), 285
isKindOf(), 285
layer(), 104
linetype(), 103
linetypeScale(), 103
list(), 105
listXrecord(), 163
mesh(), 696
moveGripPoints(), 359, 361
moveGripPointsAt(), 105
moveStretchPointsAt(), 105
new(), 85
newIterator(), 152, 154
not allowed while a DCL dialog box is active, 783
numeric conversion, 266
object snap, 252
ON_MESSAGE(), 168
OnInitDialog(), 183, 196
OnModified(), 607
open(), 328
otherInsert(), 507
otherWblock(), 505
overriding AcDbCurve functions, 292
overriding AcDbObject functions, 292
overriding AcRxObject functions, 292
passing pick points, 248
pausing for user input, 248

functions (*continued*)
pline(), 704
position(), 137
PostNcDestroy(), 184
printdxf(), 224, 226
purge(), 324
queueForGraphicsFlush(), 455
rbChain(), 161
releaseDatabaseMutex(), 668
restoreForwardingXrefSymbols(), 76
restoreOriginalXrefSymbols(), 76
return values and results for global functions, 523
returning values to AutoLISP, 265
rxInit(), 289
sampler(), 372, 375
saveComplete(), 398
setAt(), 94, 153
setAttributes(), 684, 686
setColor(), 154
setColorIndex(), 102
setDefaultFormatForSave(), 62
SetDialogName(), 183
SetDirty(), 183
setElevation(), 137
setFromRbChain(), 161
setHighlight(), 154
setLayer(), 104, 154
setLinetype(), 103, 154
setLinetypeScale(), 103
SetObjectId(), 606
setPosition(), 137
setVisibility(), 104, 154
setXData(), 86
string conversion, 266
subCancel(), 328
subClose(), 328
subentPtr(), 106, 112
subErase(), 328
subOpen(), 328
system variables, 249
text utility box, 254
traits, 355
transactionAborted(), 456
transactionEnded(), 456
transactionStarted(), 456
transformation, 363
transformBy(), 105, 107, 363, 391
unit conversion, 269
update(), 375
user input, 258
vertexPosition(), 137
viewport descriptors, 253
viewportDraw(), 105, 684, 688
visibility(), 104
wblock(), 63

functions (*continued*)
wblockClone(), 321, 468, 476, 488, 500, 502
wblockCloneObjects(), 70
workingDatabase(), 21
worldDraw(), 105, 684, 687
xData(), 86
xrefBlockId(), 76

G

geometric utilities, 253
geometry
3D, 730
basic types, 730
library, 725
parametric, 733
getAt() function, 147
getDatabaseMutex() function, 668
GetDialogName() function, 184
getEcs() function, 136
getFilerStatus() function, 300
getGeomExtents() function, 105
getGripPoints() function, 105, 359–361
getGsMarkersAtSubentPath() function, 106
getObject() function, 451, 453
for newly created objects, 454
in open mode, 453
open and close mechanism with, 455
GetObjectId() function, 607
getOsnapInfo() function, 559
getOsnapPoints() function, 105–106, 357
overriding, 357
getStretchPoints() function, 105, 361
getSubentPathsAtGsMarker() function, 106, 111
getting user input, 258
getTransformedCopy() function, 105, 107, 363
global functions, 245
global utility functions, 521
glyphs, custom, 559
goodbye() function, 397, 411
graphics and text screens
controlling, 275
low-level access, 275
reversing damage, 276
graphics generation, and transactions, 455
graphics interface library, 13, 683
Graphics System Markers, 692
example diagram, 109
graphicsModified() function, 411
grip points, 349
stretch points as subset of, 361
group codes. *See* DXF group codes
GROUP dictionary, 28, 153
group names for commands, 40

groups
adding to group dictionaries, 28
assign properties to all members, 154
GS markers. *See* Graphics System Markers

H

handlers
creating dialog, 190
handles
for DCL dialog boxes and tiles, 792
in extended data, 240
object, 21
requesting object, 83
translation, 240
handling
external applications, 532
hard references to AcDbEntities during wblockClone(), 502
hard references in long transactions, 70
header files, required, 771
help
context, 174
F1, 174
hiding DCL dialog boxes, 789
hierarchy
container classes, 145
entity classes, 99
xrecords, 242
See also class hierarchy
highlight() function, 106, 111
highlighting
nested block references, 116
subentities, 111
hyperlinks
AcDbHyperlink class, 139
associating with entities, 139
collections, 139
example, 140
nested, 139
objects, 139

I

IAcadBaseObject interface, 606
Clone() function, 607
GetClassId() function, 607
GetObjectId() function, 607
NullObjectId() function, 607
SetObjectId() function, 606
IAcadBaseObject interface interface
OnModified() function, 607
IAcDcContentBrowser interface, 634, 638
IAcDcContentFinder interface, 635
IAcDcContentFinderSite interface, 634
IAcDcContentView interface, 634, 639
IAcPostDrop interface, 635
ICategorizeProperties interface, 624
ID maps
cloning, 470
for ddp cloning (table), 492, 504
use with inserting, 504
identity matrix, 220
IdMap, 321
IdPair, 321
IDynamicProperty interface, 630–631
image creation for DCL, 801
implementing
automation objects, 610
deepClone() for custom classes, 476
DWG filing functions, 300
DXF filing functions, 302
grip point function, 359
interfaces for AutoCAD DesignCenter, 638
member functions, 297
protocol extension, 512
snap point functions, 357
static Object Property Manager interfaces, 625
stretch point functions, 361
include directory, ObjectARX, 17
indexing and filtering
block iteration, 78
defining a query, 78
interfaces provided, 77
list of classes, 77
overview, 77
processing a query, 78
querying the database, 78
registering, 77
updating, 77
initial items in a database, 60
initializing
applications, 37
classes, 289
modes and values for DCL tiles, 795
input context events, 569
input controls
user, 377
values (list), 377
input options for user input functions, 260
input point
filtering, 575
management, 567
manager class, 567
monitoring, 575
processing, 557
INSERT command, 149
cloning application, 476
inserting, 504
blocks with references into drawings, 129
databases, 65
resolving conflicts, 65
installation, modifying Windows system registry at, 47

installing ObjectARX, 16
interacting
with multiple documents, 428
with other environments, 9
interactive output, 273
interoperability of applications, 2
intersecting
for points, 107
with other entities, 369–370
with custom entities, 369
with native entities, 369
intersectWith() function, 105, 107–108, 357, 364, 369
IOPMPropertyExpander interface, 625
IOPMPropertyExtension interface, 625
IPerPropertyBrowsing interface, 624
IPropertyManager interface, 630
isolines, 710
iterating
over dictionary entries, 158
over open documents, 417
over tables, 152
through block table records, 133
through vertices in a polyline, 135
iterators, 152

K

kAllAllowedBits value, 391
kCfgMsg message, 32, 35
kColorChangeAllowed value, 390
kDependencyMsg message, 32, 34
kEdgeSubentType subentity, 111
kEndMsg message, 32, 35
kEraseAllowed value, 390
keyword comparison, 263
keyword lists, 260, 262, 375
keyword specifications, 262
keywords in user input, 259
keywords, custom object snap, 559
kFaceSubentType subentity, 111
kForNotify mode, 84
kForRead mode, 84
kForWrite mode, 84
kInitAppMsg message, 31, 33
kInvkSubrMsg message, 32, 34
kLayerChangeAllowed value, 391
kLinetypeChangeAllowed value, 391
kLinetypeScaleChangeAllowed value, 391
kLoadDwgMsg message, 31, 34
kNoDependencyMsg message, 32, 34
kNoOperation value, 390
kOleUnloadAppMsg message, 33–34
kPreQuitMsg message, 31, 34
kQuitMsg message, 32, 35
kSaveDwg message, 247
kSaveMsg message, 32, 35
kTransformAllowed value, 390
kUnloadAppMsg message, 31, 33
kUnloadDwgMsg message, 32, 34
kVisibilityChangeAllowed value, 391

L

last saved by Autodesk software, 80
LAYER command, 149
layer table, 147, 149
layers
creating, 26
database values, 67
entity, 104
layout dictionary, 156
layout manager, 160
layouts
dictionary, 160
model space, 159
ObjectARX classes, 159
paper space, 159
lengthy operations, 264
libacbr.dll file, 751
libraries
acad.lib, 10
AcBr, 751
AcDb, 12
acdb15.lib, 10
AcEd, 12
acedapi.lib, 10
AcGe, 14, 725
acge15.lib, 10
AcGi, 13, 683
acgiapi.lib, 10
AcRx, 10
acrx15.lib, 10
adui15.lib, 172
directory, ObjectARX, 17
required ObjectARX, 10
rxapi.lib, 10
search path, 43
limit of size for extended data, 239
line classes, 732
linear algebra operations, 732
linetype
entity, 102
scale, 66, 103
table, 147
values, 66
linked lists, 540
AutoLISP, 551
command and function invocation lists, 554
creating, 549
deleting, 549
dynamically allocated data, 546
entity lists with DXF codes, 553
nested, 551

linking
VC++ project settings, 169
with MFC, 168
with ObjectARX libraries, 10
LIST command, proxy entity messages with, 389
list value handling in DCL, 799
list() function, 105
lists
keyword, 262, 375
loaded applications, 43
listXrecord() functions, 163
loading
applications, 43, 53, 772
demand, 45
localization for ObjectDBX, 661
locking
documents, 417
documents for Automation requests, 612
explicit document locking, 425
logo compliance, 2
logo program for ObjectARX, 2
long transactions
controlling, 70
deep cloning, 70
example, 71
hard references, 70
notifications, 70
overview, 69
read-only access, 70
starting, 70
tracking, 69
LongTransaction issues for custom objects, 321
LongTransactionManager, 321
lookup order, commands, 42
LTSCALE system variable, 66

M

macros
AC_DECLARE_EXTENSION_MODULE, 171
AC_IMPLEMENT_EXTENSION_MODULE, 171, 187
ACRX_DECLARE_MEMBERS, 286
ACRX_DXF_DEFINE_MEMBERS, 388, 390
ACRX_NO_CONS_DEFINE_MEMBERS, 287
ACRX_X_CALL, 515
actrTransactionManager, 451
ads_point_set(), 534
cast(), 285
class implementation, 287
custom class function declaration, 286
desc(), 285
isA(), 285
isKindof(), 285
managing applications with Windows system registry, 52
masking. *See* filtering
MATCH command
protocol extension for, 516
MATCHPROP command, 370
matrix
classes, 730
composition, 213
identity, 220
multiplication, 220
transformation, 535
MCS. *See* model coordinate system
MDI. *See* multiple document interface
measuring text strings, 255
memory management for extended data, 239
memory requirements for global functions, 523
merging databases, 65
mesh primitives, 696
face and edge visibility, 699
traversers, 760
mesh() function, 696
message map example, 168
messages
application reactions to AutoCAD messages, 33
kCfgMsg, 32, 35
kDependencyMsg, 32, 34
kEndMsg, 32, 35
kInitAppMsg, 31, 33
kInvkSubrMsg, 32, 34
kLoadDwgMsg, 31, 34
kNoDependencyMsg, 32, 34
kOleUnloadAppMsg, 33–34
kPreQuitMsg, 31, 34
kQuitMsg, 32, 35
kSaveMsg, 32, 35, 247
kUnloadApMsg, 33
kUnloadAppMsg, 31
kUnloadDwgMsg, 32, 34
only responded to by applications using ActiveX, 33
responding to AutoCAD, 31
sent only if AutoLISP function registered, 32
sequence of, 35
to applications that have registered services, 32
WM_ACAD_KEEPFOCUS, 168
MFC. *See* Microsoft Foundation Classes
Microsoft Component Object Model (COM). *See* COM
Microsoft Foundation Classes
access to AutoCAD ActiveX Automation, 595
and modeless dialog boxes, 168
debugging ObjectARX applications, 169
dynamic linking, 168–169
modeless dialog boxes, 168

Microsoft Foundation Classes (*continued*)
resource management, 170
topics, 167
user interface support, 172
user-interface support, 172
Microsoft Visual C++ project settings
for dynamically linked MFC, 169
using AdUi and AcUi with AppWizard, 186
migrating ADS Programs to ObjectARX, 769
acrxEntryPoint() function, 770
building ADS applications for ObjectARX, 773
example, 773
header files, 771
loading applications, 772
MIRROR command, cloning application, 476
mixing the transaction model with the open and close mechanism, 455
Mline style dictionary, 156
modal commands, 42
model coordinate system (MCS), 217, 711
model coordinates, transforming to world coordinates, 217
model space
adding entities to, 23
layouts, 159
versus paper space, 23
modeless dialog boxes
focus, 168
MFC, 168
modes
for opening objects, 84
kForNotify, 84
kForRead, 84
kForWrite, 84
modified() function, 394, 397, 411
modifiedXData() function, 397, 411
modifying Windows system registry at installation, 47
modifyUndone() function, 397, 411
monitoring input points, 575
most recently used combo boxes, 179
moveGripPoints() function, 359, 361
moveGripPointsAt() function, 105, 360–361
signature, 359
moveStretchPointsAt() function, 105, 361
multi-document commands, 432
multiple databases, 21
multiple document interface (MDI), 415
application-specific data, 431
compatibility levels, 423
database undo, 438
MDI-Aware compatibility, 424
MDI-Capable compatibility, 427
MDI-Enhanced compatibility, 427
minimum requirements, 423
multiple document interface (MDI) (*continued*)
SDI-Only compatibility, 423
terminology, 418
transaction management, 438
multiple documents, interacting, 428

N

named object dictionary, 144, 241, 480
defaults, 22
names
global versus local command names, 42
symbol table records and dictionaries, 147
symbol tables, 243
nested
block reference highlighting, 116
DCL dialog boxes, 789
entities, 221
hyperlinks, 139
transactions, 451
newIterator() function, 152
nonreentrant commands, 432
notification
custom, 398
document events, 430
for AcDbObject and database, 398
functions, 394
immediate, 410
immediate versus commit-time events, 410
in long transactions, 70
object reactors for, 402
obtaining ID of, 402
types of, 395
overview, 394
reactor classes for, 394
reactor lists, 394
reactors for, 396
use guidelines, 412
using, 394
numActiveTransactions() function, 412
numbers, real, 534
numeric conversions, 266
examples, 268

O

object IDs
commit-time and, 454
for named object dictionaries, 480
obtaining, 21
obtaining object pointers from, 450, 453
ownership, for cloning, 469
object pointers
in nesting transactions, 451
obtaining in transactions, 450, 453
with getObject() functions, 453

Object Property Manager API, 593, 622
adding properties for OPM, 626
AutoCAD COM Implementation, 623
categorizing properties for OPM, 626
dynamic properties, 630
ICategorizeProperties interface, 624
IDynamicProperty interface, 630–631
implementing static OPM interfaces, 625
IOPMPropertyExpander interface, 625
IOPMPropertyExtension interface, 625
IPerPropertyBrowsing interface, 624
IPropertyManager interface, 630
static COM interfaces, 624
object snap, 252
object snap manager, 558
object snap points, 106
for intrinsic entity functions, 357
objectAppended() function, 399
ObjectARX
application examples, 773
application loading, 772
common characteristics of global functions, 522
communication between applications, 528
comparison of global function calls to AutoLISP calls, 522
creating objects in, 25
creating the MFC application skeleton, 186
defining AutoLISP functions, 524
deriving custom classes, 284
directory tree, 16
DXF group codes, 542
dynamically allocated linked lists of result buffers, 546
entity names, 538
error handling for global functions, 527
extension dictionary example, 89
extension dictionary global functions example, 91
external application handling, 532
global function argument lists, 522
global function memory requirements, 523
global function return values, 523
global utility functions, 521
include directory, 17
installing, 16
library directory, 17
logo program, 2
macros
object reactor classes and, 402
reactor classes and, 397
opening and closing objects, 27
overview, 7
result buffers, 540
result type codes, 541
samples directory, 17
selection set names, 538
ObjectARX (*continued*)
types, 533
user input control bit codes, 545
user interface, 172
using MFC with applications, 168
values, 533
variables, 533
wizard, 30
ObjectARX-exclusive data type, 778
objectClosed() function, 397, 412
ObjectDBX, 655
AcDbHostApplicationServices class, 658
AcDbTransactionManager class, 663
AcEditorReactor notifications, 660
AcGi API, 664
AcGix API, 665
AcGixSimpleView class, 667
AcGixWhipView class, 668
active viewports in Model Space, 678
AcTransaction class, 663
AcTransactionManager class, 663
AcTransactionReactor class, 663
application services class, 658
creating a viewer, 663
demand loading, 671
differences from ObjectARX, 659
DWG files from earlier releases, 680
extended entity data (EED or xdata), 681
getDatabaseMutex() function, 668
getting started and using, 657
host application, 656
insert() function, 678
installing the libraries for your application, 672
libraries, 656
localization, 661
overview, 656
raster images, 682
releaseDatabaseMutex() function, 668
representing TrueType fonts in 3D space, 666
SimpleView, 667
tips and techniques, 676
transaction management, 663
TrueType font elaboration, 666
user interface and database access, 657
using the database mutex, 668
ViewAcDb viewer, 669
viewports, 679
WhipVIew, 668
XMX files, 661
objectErased() function, 399
objectModified() function, 399
objects
AcApDocManager, 417
AcApDocument, 417
AcDbDictionary, 144

objects (*continued*)
AcDbHyperlink, 139
AcDbLayout, 156
AcDbObject, 82
application-specific document objects, 431
closing, 82
container, 143
creating in AutoCAD, 22
creating in ObjectARX, 25
creating of protocol extension classes, 514
deleting, 85
document, 417
erasing, 94
error codes during opening, 84
exploding, cloning and, 477
filing, 95
handles, 21
implementation of automation, 610
newly created, and transactions, 454
obtaining pointers to, in transactions, 453
open modes, 84
opening, 82
opening and closing process diagram, 82
overview, 82
ownership, 85
proxies, 387
references, 310
relationships between, 469
requesting handles, 83
snap modes, 106
snap modes, custom, 558
snap points, 106
solid, 757
using drawables in objects, 708
version support, 343
xrecord, 241
object-specific data
adding, 86
objList space objects, 472
obtaining object IDs, 21
ol_errno.h file, 771
ON_MESSAGE() function, 168
OnInitDialog() function, 183, 196
online documentation, 2
OnModified() function, 607
open() function, 328
openedForModify() function, 397, 410–411
opening ObjectARX objects, 27
operations
databases, 59
linear algebra, 732
wblock, 63
OPM. *See* Object Property Manager API
order dependence, 304
organization of extended data, 235
origin point of text, 254
otherInsert() function, 507
otherWblock() function, 505
overriding
AcDbEntity functions, 350–352
common entity functions, 353
getOsnapPoints() function, 357
resources, 171
saveAs() function, 355
viewportDraw() function, 354
worldDraw() function, 350, 353–354
overview of ObjectARX, 7
ownership
and cloning, 469, 489
building a hierarchy, 312
cloning objects from different owners, 472
entities, 99
hard, 312, 469, 488
in named object dictionaries, 480
objects, 85
of entities, 242
references, 311
soft, 312, 469
structure for database entities, 99
types of cloning and, 469

P

paper space
layouts, 159
linetype scale, 104
versus model space, 23
paper space display coordinate system
definition, 272
parametric geometry, 733
partial undo, 325
passing arguments in DCL callback functions, 786
paths
library search, 43
subentity, 110
pause symbol, 248
pausing for user input in commands, 248
PDB. *See* programmable dialog boxes
per-document data, 424
persistency
of AcGe entities, 746
of dialog data, 182
pick points
example in commands, 249
passing to commands, 248
picking
forced entity, 568
plane classes, 732
pline() function, 704
plot settings, 160
point classes, 730
pointers
AcEdJig and, 371
hard, 320, 469, 488
illustrated, 490

pointers (*continued*)
obtaining pointers to objects in transactions, 453
soft, 321, 469
points, 534
polygonDc() functions, 355
polygonEye() functions, 355
polygons
primitives for drawing, 355
stretching, 360
polyline primitive, 704
polylineDc() functions, 355
polylineEye() functions, 355
polylines
primitives for drawing, 355
scaling AsdkPoly and, 363
populating databases, 60
position() function, 137
PostNcDestroy() function, 184
pOwner parameter, 489
primitives, 696
arc, 703
defined, 354
functions for drawing, 354–355
mesh, 696
polyline, 704
shell, 700
text, 704
printdxf() function, 224, 226
printed manuals, 2
printing text on screen, 273
profile manager, 586
programmable dialog boxes, 779
prompt line input, 259
prompting for input, 258
properties, common entity, 101, 691
protecting application extended data, 239
protocol extension, 511
class descriptor objects, 514
structure, 514
defining classes, 512
for custom object snaps, 559
implementing, 512
default class for, 515
for MATCH command, 516
unloading the application for, 515
MATCH command, 516
registering classes of, 513
using in applications, 515
proxy objects
for entities and objects, 387
creating, 389
custom objects and, 388–389, 391
displaying proxy entities, 390
editing proxy entities, 390
proxy objects (*continued*)
entities
displaying, 390
editing, 390
modification of, 388
object life cycle, 388
unloading application for, 391
user encounters with, 389
using proxy objects, 389
PROXYGRAPHICS system variable, 390
PROXYSHOW system variable, 390
PSLTSCALE system variable, 66, 104
purge() function, 324
purging entities, 223, 324

Q

queueForGraphicsFlush() function, 455
quiescent, definition, 422

R

raster images in ObjectDBX, 682
rbChain() function, 161
RDS. *See* registered developer symbol
reactors, 394, 456, 507
AcDbObject and database notification events, 398
classes, 394
custom notifications and, 398
database, using, 394
document, 418
editor
deep cloning and, 467
using, 394
lists, 394
object, 395
building in object dependencies with (example), 404
ending transactions, 412
immediate versus commit-time events, 410
modifying, 411
obtaining ID of, 402
persistent, 395, 402
read-only state of, 411
transient, 395, 402
types of, 395
using, 394, 402
ObjectARX macros and, 397
obtaining the ObjectID, 402
persistent, 395
deriving, 402
transaction, 456
using, 394, 396
reading a drawing example, 67

read-only access in long transactions, 70
real numbers, 534
reappended() function, 397–398, 411
recovering deleted entities, 216
redo, 324
redrawing the graphics screen, 275
references
    handling hard references to AcDbEntities during wblockClone(), 502
    objects, 310
    ownership, 311
    pointer, 320
regapp table, 147
regen type. *See* regeneration type
regenerating
    drawings, 104, 684
    entities, 234
regeneration type, viewport, 689
registered developer symbol (RDS), 3, 40–41, 144
registering
    application names, 237
    as MDI-Aware, 427
    new commands, 40
    protocol extension classes, 513
registry. *See* Windows system registry
relational tests in filtering, 207
relationship between DuplicateRecordCloning and DeepCloneType values, 71
relationship between symbol table and deep clone types, 71
releaseDatabaseMutex() function, 668
reloading external references, 76
removing Windows system registry information, 49
renaming classes, 346
requesting, object handles, 83
requirements
    ObjectARX libraries, 10
    software and hardware, 16
resbuf, 540, 546
    example, 204–205
    value types, 204
resource management
    detaching resources, 187
    explicit setting, 170
    overriding resources, 171
    overview, 170
    switching resources, 170
responding to AutoCAD messages, 31
restoring color or linetype, 227
restoring state, 326
restype, 541
result buffers, 540, 546
result type codes, 541
retrieving blocks example, 242
retrieving data interactively, 258
retrieving definition data example, 227
retrieving extended data, 238
retrieving extended data for specific applications example, 238
retrieving user input, 258
returning values to AutoLISP functions, 265
RSERR, 529
RSRSLT, 529
RT codes, 541
RTCAN, 259
RTERROR, 259
RTKWORD, 259
RTNONE, 259
RTNORM, 259, 523, 532
RTREJ, 259
running applications from AutoLISP, 55
runtime
    class identification, 285
    type identification mechanism, 11, 283
rxapi.lib library, 10
rxboiler.h file, 284
rxdefs.h file, 771
rxInit() function, 289

S

sampler() function, 372
    with drag loop, 372
    implementing, 375, 378
    obtaining angle, distance, or point with, 377
    overriding, 371
samples. *See* examples
samples directory, ObjectARX, 17
saveAs() function
    overriding, 355–356
    proxy entities and, 390
saveComplete() function, 398
saving
    considerations, 62
    databases, 61
    entities, 356
    file formats for drawings, 61
    global functions, 63
saving a drawing, example, 67
scaling
    interactive, 212
    selection sets, 211
screen input, 259
    low-level access, 275
scrolling tabs, 183
SDI system variable, 422
SDI. *See* single drawing interface
search for files by name, 251

selecting extended data, 235
selection sets
creating, 200
deleting items, 209
error handling, 214
filtering, 203
freeing, 203
graphically dragging, 263
interactive scaling, 212
interactive transformation, 212
limitations, 203
manipulating, 209
name, 200
name volatility, 200
names, 538
options, 201
scaling, 211
transformation, 211
sentinel code
extended data, 235
resbufs, 226
sequence of events in applications, 35
setAt() function, 94, 153
setAttributes() function, 684, 686
setColor() function, 154
SetDialogName() function, 183
SetDirty() member function, 183
setDispPrompt() function, 372, 375
setFromRbChain() function, 161
setHighlight() function, 154
setKeywordList() function, 372, 375
setLayer() function, 154
setLinetype() function, 154
SetObjectId() function, 606
setPosition() function, 137
setSpecialCursorType() function, 372, 375
setting
current database values, 66
current document without activating, 430
database color values, 66
database linetype scale values, 66
database linetype values, 66
default file format, 61
setting up DCL list boxes and pop-up lists, 797
setUserInputControls() function, 372, 377
setVisibility() function, 154
setXData() function, 86
sheep cloning. *See* deep cloning
shell primitive, 700
SimpleView, 667
single drawing interface (SDI), 422
compatibility with MDI, 423
single-screen AutoCAD installations, 275
size limits for extended data, 239
skeletons
creating, 186
snap modes, 106
custom objects, 558
keywords, 559
snap points, 106
implementing, 357
solid objects, 757
specifying a coordinate system, 271
stack, command, 40
stacked tabs, 183
starting long transactions, 70
storing information in objects, 242
STRETCH command, 361
stretch point functions, 361
stretch points, 349, 361
in grip editing, 359
string comparisons
ignoring case, 279
with wild-cards, 278
string conversions, 266
examples, 267
leading and trailing zeros, 267
precision, 267
units used, 267
string globalization, 555
struct resbuf, 540
subCancel() function, 328
subClose() function, 328, 398
subentities
highlighting, 111
kEdgeSubentType, 111
kFaceSubentType, 111
paths, 110
subentPtr() function, 106, 112
subErase() function, 328
subObjModified() function, 397, 411
subOpen() function, 328
subsidiary functions, 328
summary information for drawings, 79
supporting DCL dialog boxes in an application, 780
surfaces, AcGe functions, 733
switching, disabling document, 435
symbol tables, 144
accessing, 242
comparison with dictionaries, 144
contents at startup, 60
defaults, 22
getAt() function, 147
interating
over symbol tables, 152
names, 243
names in records, 147
VPORT, 243
system cursor, disabling, 568
system registry. *See* Windows system registry
system requirements, 16

system variables
accessing, 249
APERTURE, 252
AUNITS, 267
AUPREC, 267
CECOLOR, 66
CELTSCALE, 66, 103
CELTYPE, 102
CMDACT, 569–570
CPROFILE, 586
DEMANDLOAD, 49
DIMZIN, 267
example of accessing, 249
freeing string space, 250
LTSCALE, 66
LUNITS, 267
LUPREC, 267
PROXYGRAPHICS, 390
PROXYSHOW, 390
PSLTSCALE, 66, 104
SDI, 422
TABMODE, 276
UNITMODE, 267, 269
sysvars. *See* system variables

T

tab dialogs
constructing, 183
extending, 183
extending built-in, 184
extensibility, 176
scrolling tabs, 183
stacked tabs, 183
using, 183
tables
AcApProfileManager class capabilities, 586
AcApProfileManagerReactor class notifications, 587
AcGe global identifiers and header files, 728
application reactions to AutoCAD messages, 33
arbitrary user input, 262
ATL-based templates, 611
character type functions, 270
command lock types, 426
CursorType values, 376
DCL ads_start_dialog() status code values, 793
DCL callback reason values, 788, 794
DCL color numbers, 801
DCL list function code values, 794
DCL tile mode values, 795–796
deep clone types and duplicate record cloning, 71
DXF group code ranges for object representation, 303
tables (*continued*)
DXF group coderanges for xrecords, 162
entity colors, 101
error codes, when opening objects, 84
exploding entities, 123
IAcDcContentBrowser interface functions, 638
IAcDcContentView interface functions, 639
ID map for overriding the wblockClone() function examples, 492
input context events, 569
input options set by acedInitGet(), 260
library function result type codes, 544
mesh traversers, 761
message handlers, 191
messages sent to ObjectARX applications, 31–33
object snap modes, 106
ObjectDBX libraries, 658
proxy flags options, 390
result type codes, 541
return values for user-input functions, 259
save format, 61
SDI system variable values, 422
selection set options for acedSSGet(), 201
states for opening objects, 303
topological traversers, 759
user-input function summary, 258
value-return function summary, 265
XMX file types, 661
tablet calibration, 276
example, 276
normalization, 278
transformation matrix, 277
tessellation, 709
text bounding boxes, 254
text box utility, 254
text origin, 254
text primitive, 704
text strings
globalization, 555
measurement, 255
text style table, 147
TextTips, 174
tiles in DCL, 795
tip windows, 174
tolerances, 729
ToolTip string, custom object snaps, 559
ToolTips windows, 174
topological objects, 756
brep, 756
complex, 756
edge, 756
face, 756
loop, 756
shell, 756
vertex, 756

topological traversers, 758
topTransaction() function, 451
traits
entity, 690
subentity, 691
transaction management
for ObjectDBX, 663
in MDI environment, 438
transaction model
commit-time guidelines in, 454
graphics generation and, 450, 455
mixing, with open and close mechanism, 450
newly created objects and, 454
obtaining pointers to objects in, 450, 453
reactors, 456
stack, 452
undo mechanism and, 450
transactionAborted() function, 456
transactionEnded() function, 412, 455–456
transactions
aborting or ending, 452
boundaries, 452
graphics generation, 455
handling newly created objects, 454
management, 449–450
manager, 451
model, 450
nested, 451
obtaining pointers to objects, 453
reactors, 456
undo, 455
transactionStarted() function, 456
transformation matrices, 535
transformation, coordinate, 710
transformBy() function, 105, 107, 363, 391
applying transform matrix to entity, 363, 391
for stretch mode, 359
transforming
coordinate data, 217
coordinate systems, 271
functions, 107, 363
interactive, 212
selection sets, 211
transient reactors, 395
deriving, 402
translating
AcDbObjectIds and ads_names, 83
and cloning, 470
transparent commands, 42
traversers
mesh, 760
topological, 758
TrueType font elaboration for ObjectDBX, 666
types
ads_point, 271

U

UCS. *See* user coordinate system
UCS table, 147
unappended() function, 397–398, 411
undo, 324
automatic, 325
in MDI environment, 438
partial, 325
transactions, 455
undrawing, 275
unit conversions, 269
example, 269
unloading applications, 38, 44, 53, 391, 515
unlocking applications, 44
update() function, 373
implementing, 375, 378
modifying entity with, 378
overriding, 371
user breaks, 264
user coordinate system (UCS), 271
user input
arbitrary input example, 261
AutoLISP expressions, 259
breaks, 264
cancel, 264
cancel example, 264
control bit codes, 545
controlling function conditions, 260
controls, 377
during lengthy operations, 264
from prompt line, 259
from the screen, 259
function return values, 259
functions, 258
ignoring breaks and cancels, 264
input options, 260
keywords, 259
low-level access, 275
pausing for in commands, 248
raw, 275
restricted input example, 261
retrieving, 258
user interface
in ObjectARX, 172
initialization, 172
support, 172

V

value types
resbufs, 204
values
attributes, 355
cursor (table), 376
points, 534
user input controls (list), 377
vector classes, 730

Veritest, 2
version support
class data, 347
class xdata, 347
classes, 343
objects, 343
using class versioning, 344
vertexPosition() function, 137
vertices in a polyline, iterating through, 135
view table, 147
ViewAcDb viewer, 669
viewers, 663
viewport descriptors, 253
viewport regeneration type, 689
viewport table, 147
viewportDraw() function, 105, 353, 684, 688
proxy entities and, 390
regenerating graphics with, 354
saving and, 356
view specificity with, 355
view-dependence with, 354
visibility entity, 104
visibility, mesh face and edge, 699
Visual C++. *See* Microsoft Visual C++
VPORT symbol table, 243
vports AutoLISP function, 253

W

watch_db() function, 399
wblock
cloning and filing, 469
cloning and ownership, 469
notification functions, 505
operations, 63
WBLOCK command, 507
cloning application, 477, 480
wblockClone() function, 321, 488, 500, 502
AutoCAD commands using, 476
deep cloning with, 467
pOwner parameter, 489
symbol table record and, 489
translation phase, 477
WCS. *See* World Coordinate System
WhipView library, 668
wild-card
character matching, 278
ignore case, 279
matching example, 278
wild-card (*continued*)
patterns, 278
patterns in filters, 205
special characters, 278
Windows system registry
application entries, 46
creating file for custom object COM server, 613
keys and values, creating at installation, 48
managing applications with, 52
modifying at application installation, 47
removing information, 49
requirements for AutoCAD DesignCenter API, 635
subkeys and values, creating in AutoCAD, 48
user profiles, 182
wizard, ObjectARX, 30
WM_ACAD_KEEPFOCUS, 168
workingDatabase() function, 21
World Coordinate System (WCS), 217, 712
definition, 271
worldDraw() function, 105, 684, 687
custom entities displayed in, 353
overriding, 350, 353
proxy entities and, 390
regenerating graphics with, 354, 373
saving and, 356
supporting proxy entity graphics, 356
view-independence with, 354
write, open and close mechanism for, 450

X

xdata, 86
ads_binary structure, 554
class version numbers, 347
exclusive data types, 554
for ObjectDBX applications, 681
See also extended data
xData() function, 86
XMX files for ObjectDBX, 661
xrecords, 161
DXF group codes, 241
example, 163
hierarchy, 242
objects, 241
versus extended data, 242
xrefs. *See* external references