Third Party Software Program Credits

AutoCAD Release 13 incorporates licensed, run-time versions of 386|DOS-Extender and 386|VMM from Phar Lap Software, Inc., Cambridge, Massachusetts.

The installation program used to install AutoCAD Release 13, INSTALL, is licensed software provided by Knowledge Dynamics Corporation, Highway Contract 4 Box 185-H, Canyon Lake, Texas 78133-3508 (USA), 1-512-964-3994. INSTALL is Copyright © 1987-1989 by Knowledge Dynamics Corporation, which reserves all copyright protection worldwide. INSTALL is provided to you for the exclusive purpose of installing AutoCAD Release 13.

ACIS ® is a registered trademark of SPATIAL TECHNOLOGY, INC.


Portions licensed from D-Cubed Ltd. DCM_2D is a trademark of D-Cubed Ltd. DCM_2D Copyright © D-Cubed Ltd. 1989-1995

International CorrectSpell™ Spelling Correction System © 1993 by INSO Corporation. All Rights Reserved. Reproduction or disassembly of embodied algorithms or database prohibited.

InstallSHIELD™ 3.0. Copyright © 1995 Stirling Technologies, Inc. All Rights Reserved.


Typefaces from the Bitstream ® typeface library.

The license management portion of this product is based on

Élan License Manager
All rights reserved.
CHAPTER 2 ■ DATABASE PRIMER  21
AutoCAD Database  22
   Multiple Databases  23
   Object ID  23
Essential Database Objects  24
   Creating Objects  24
   Sample ARX Code  26
      Creating Entities  26
      Creating a New Layer  27
      Opening and Closing Objects  28
      Adding a Group to the Group Dictionary  29

CHAPTER 3 ■ WRITING AN ARX APPLICATION  31
ARX Application Structure  32
   AutoCAD Messages to ARX Applications  33
   Sequence of Events in an ARX Application  34
Registering New Commands  36
   Command Stack  37
   Lookup Order  38
      Global versus Local Command Names  38
      Transparent Commands  39
      Querying Commands  40
Example Application  40
Loading an ARX Application  41
Unloading an ARX Application  41
ARX Command  43
Error Handling  45
Memory Management  48

CHAPTER 4 ■ DATABASE OPERATIONS  49
Initial Database  50
Creating and Populating a Database  50
Saving a Database  51
The wblock Operation  51
   Creating a New Database from an Existing Database  51
   Creating a New Database with Entities  52
      Copying a Named Block  52
      Copying an Array of Entities  52
Inserting a Database  52
Complex Entities 111
  Creating a Complex Entity 111
  Iterating Through Vertices in a Polyline 112
Coordinate System Access 113
  Entity Coordinate System 113
  AcDb2dPolylineVertex 114
Curve Functions 114

CHAPTER 7 ■ CONTAINER OBJECTS 117
Comparison of Symbol Tables and Dictionaries 118
Symbol Tables 120
  Block Table 122
  Layer Table 123
    Layer Properties 123
    Creating and Modifying a Layer Table Record 124
  Iterators 125
    Iterating over Tables 126
Dictionaries 127
  Groups and the Group Dictionary 127
  Mline Style Dictionary 130
  Creating a Dictionary 130
  Iterating over Dictionary Entries 131
Xrecords 132
  DXF Group Codes for Xrecords 133
  Examples 134

PART II DEFINING NEW CLASSES 139

CHAPTER 8 ■ DERIVING A CUSTOM ARX CLASS 141
Run-Time Class Identification 143
Class Declaration Macro 143
Class Implementation Macros 144
Class Initialization Function 146

CHAPTER 9 ■ DERIVING FROM ACDBOBJECT 147
Overview 148
  AcDbObject: Essential Functions to Override 148
  AcDbObject: Functions Often Overridden 148
  AcDbObject: Functions Sometimes Overridden 149
  AcDbObject: Functions Rarely Overridden 149
  AcRxObject: Functions Rarely Overridden 150
  AcDbEntity: Functions to Override 150
  AcDbEntity: Functions Usually Overridden 151
Unloading an Application 198
Summary of Zombie Behavior 198
DXFIN 198
DXFOUT 198
OPEN 199
SAVE 199
Object Version Support 199

CHAPTER 10  DERIVING FROM ACDBENTITY 201
Displaying the Entity 202
OVERRIDING AcDbEntity::saveAs() 204
Scope of AcGi Objects 205
Regeneration Type 205
Setting Entity Traits 205
Subentity Traits 206
Fill Type 207
GS Markers 207
Useful AcGi Constants 208
Example 208
Primitives 211
Mesh 211
Shell 215
Arc 218
Text 219
Converting a Text Style Record into an AcGi Primitive 221
Tessellation 222
Isolines 223
Viewports 223
Transformations 223
Example 1: Coordinate Systems 225
Example 2: Determining Hidden Lines for an Object for Standard Display 226
Example 3: Obtaining the Window Coordinates 230
Example 4: Calculating the Circle to Draw 231
Common Entity Functions 234
Object Snap Points 234
Grip Points 236
Stretch Points 239
Transformation Functions 241
Intersecting with Other Entities 242
Intersecting a Custom Entity with Another Entity 247
Using AcEdJig 248
Deriving a New Class from AcEdJig 248
General Steps for Using AcEdJig 248
CHAPTER 13  DEEP CLONING  299
Deep Clone Basics  300
   Using clone() versus deepClone()  300
Key Concepts  300
   Filing  300
   Ownership  301
   ID Map  301
   Cloning and Translating  301
Typical Deep Clone Operation  302
Cloning Objects from Different Owners  304
Implementing deepClone() for Custom Classes  306
   AutoCAD Commands that Use Deep Clone and Wblock Clone  307
Cloning Phase  307
Translation Phase  308
   Case 1  308
   Case 2  309
   Case 3  309
Named Object Dictionary  310
Overriding the deepClone() Function  314
Overriding the wblockClone() Function  318
Insert  325
Editor Reactor Notification Functions  325

CHAPTER 14  PROTOCOL EXTENSION  331
Defining Protocol Extensions  332
Registering Protocol Extensions  333
   Default Class for Protocol Extension  334
Unloading the Application  334
Using Protocol Extensions in an Application  335
Protocol Extension Example  335
CHAPTER 15  ■ USING MFC WITH AN ARX APPLICATION  341

General Guidelines  342
Creating a Project  342
Initializing MFC  343
Other DLL Initialization  343
The Application Object  344
Memory Allocation  344
Modeless Dialogs  345
Using AutoCAD’s Frame and View Window Handle  346
Example  346
  Main Header File (MFCTEMPL.H)  346
  Application File (MFCTEMPL.CPP)  348
  Main ARX File (RXTEMPLT.CPP)  350

PART IV  APPENDIXES  351

APPENDIX A  ■ MIGRATING ADS PROGRAMS TO ARX  353

Reasons for Migrating to ARX  354
  acrxEntryPoint  354
  Header Files  355
Loading Applications — ADS versus ARX  356
Building Applications in the ARX Program Environment  357
Sample Application  357
ARX-Exclusive Data Type  361

APPENDIX B  ■ USING THE GEOMETRY LIBRARY  363

Overview  364
  Global Data and Functions  366
  Tolerances  367
Using Basic Geometry Types  368
Using the Line and Plane Classes  369
Parametric Geometry  371
  Curves  371
    Characteristics  371
    Degeneracy  373
  Surfaces  373
Special Evaluation Classes  374
  Tips for Efficient use of Curve and Surface Evaluators  381
The AutoCAD Run-Time Extension (ARX) programming environment includes a number of C++ libraries that allow you to develop AutoCAD applications, to extend AutoCAD classes and protocol, and to create new AutoCAD commands that operate in the same way as built-in AutoCAD commands.

This document describes the ARX programming environment as implemented in the AutoCAD® Release 13 ARX portion of the AutoCAD Software Development Kit (SDK). Topics include

- Using ARX
- Defining new classes
- Specialized topics

For information on the AutoCAD Run-Time Extension (ARX) class libraries, see the ARX Reference Manual.
Using SDK Documentation


Typographical Conventions

To orient you to AutoCAD features as they appear on the screen, specific terms are set in typefaces that distinguish them from the body text. Throughout AutoCAD documentation, the following conventions are used.

<table>
<thead>
<tr>
<th>Text element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu items, prompts, and other text displayed on screen</td>
<td>From the File menu, choose Open Command: area &lt;First point&gt;/Object/Add/Subtract:</td>
</tr>
<tr>
<td>Text you enter</td>
<td>At the Command prompt, enter shape</td>
</tr>
<tr>
<td>Text you enter</td>
<td>At the Command prompt, enter shape</td>
</tr>
<tr>
<td>File names, names of directories, and instructions after prompt sequences</td>
<td>disc.dwg c:\ acad\ support Select objects: Use an object selection method</td>
</tr>
<tr>
<td>AutoCAD-named objects, such as layers, linetypes, and styles, and AutoCAD commands and system variables</td>
<td>DISC-FRONT, DASHDOT, STANDARD SAVE, COPY, DWGNAME, BLIPMODE</td>
</tr>
<tr>
<td>AutoLISP and C variables, sample code, and text in ASCII files</td>
<td>The variable pi is preset to a value of pi ***POP1</td>
</tr>
<tr>
<td>AutoLISP, ADS, ARX and DIESEL function names</td>
<td>command ads_command( )</td>
</tr>
<tr>
<td>Formal arguments specified in function definitions</td>
<td>The string and mode arguments</td>
</tr>
</tbody>
</table>
OVERVIEW

The AutoCAD Run-Time Extension (ARX) programming environment includes a number of C++ libraries that allow you to develop AutoCAD applications, to extend AutoCAD classes and protocol, and to create new AutoCAD commands that operate in the same way as built-in AutoCAD commands.

An ARX application is a dynamic link library (DLL) that shares AutoCAD’s address space and makes direct function calls to AutoCAD. Designed with extensibility in mind, the ARX libraries include macros to facilitate defining new classes and offer the ability to add functionality to existing classes in the library at run time. The ARX libraries can be used in conjunction with the AutoCAD Development System (ADS) and the AutoLISP application programming interfaces.

This chapter provides an overview of the ARX libraries.
ARX Programming Environment

The programming environment provides an object-oriented C++ application programming interface that enables developers to use, customize, and extend AutoCAD. The ARX 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 the AutoLISP and AutoCAD Development System (ADS) application programming interfaces so that developers can choose the programming tools best suited to their needs and experience.

ARX Libraries

The ARX environment consists of the following libraries:

- **AcRx**: Classes used for binding an application and for runtime class registration and identification
- **AcEd**: Classes for registering native commands and for system event notification
- **AcDb**: AutoCAD database classes
- **AcGi**: Graphics interface for rendering AutoCAD entities
- **AcGe**: Utility library for common linear algebra and geometric objects
- **ADS**: A C library used to create AutoCAD applications. ARX applications typically use this library for such operations as entity selection, selection set manipulation, and data acquisition.

For more information on the ADS library, see the ADS Developer’s Guide.
AcRx Library

The AcRx library provides system-level classes for dynamic link library (DLL) initialization and linking and for run-time class registration and identification. The base class of this library is AcRxObject, which provides facilities for

- Object run-time class identification and inheritance analysis
- Run-time addition of new protocol to an existing class (see chapter 14, “Protocol Extension”)
- Object equality and comparison testing
- Object copy

The AcRx library also provides a set of C++ macros to help you create new ARX classes, which are derived from AcRxObject (see chapter 8, “Defining a Custom ARX 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 shown in the following figure (abstract classes are shown in boldface). For more information on using AcRx classes, see chapter 3, “Building an ARX Application.”

```
AcRxObject
  AcRxClass
  AcRxDictionary
  AcRxDLinkerReactor
  AcRxDynamicLinker
  AcRxIterator
    AcRxDictionaryIterator
  AcRxService
```

AcRx class hierarchy
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. Another important class in this library is AcEditorReactor, which monitors the state of the AutoCAD editor and notifies the application when specified events occur, such as the starting, ending, or cancelling of a command.

The class hierarchy for the AcEd library is shown in the following figure (abstract classes are shown in boldface). For more information on registering new AutoCAD commands using ARX, see chapter 3, “Building an ARX Application.” For an example of using an editor reactor, see chapter 11, “Notification.”

AcRxObject
- AcEdCommandIterator
- AcEdCommandStack
- AcEditor
- AcEdJig
- AcEditorReactor
- AcTransaction
- AcTransactionManager
- AcTransactionReactor

AcEd class hierarchy

AcDb Library

The AcDb library provides classes that allow you to access the structures in 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, 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 as well as 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 properties.

- A named object dictionary (of class AcDbDictionary), which provides a primary “table of contents” for an AutoCAD drawing. Initially, this table of contents contains only two items: the IDs of the two other dictionaries used by AutoCAD. Applications, however, are free to add other objects to the dictionary.
A fixed set of about 200 header variables defined by AutoCAD. (These variables are not database objects.)

The class hierarchy for the AcDb library is shown in the following figure (abstract classes are shown in boldface). 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 9, “Deriving from AcDbObject,” and chapter 10, “Deriving from AcDbEntity.”

```
AcDbAuditInfo (Auditing and drawing recovery)
AcDbRecover
AcDbRecoverCallBack

AcDbHandle (Object identification)
AcDbObjectId
   AcDbHardOwnershipId
   AcDbHardPointerId
   AcDbSoftOwnershipId
   AcDbSoftPointerId

AcDbIdMapping (Used for deep clone)
AcDbIdMappingIter
AcDbIdPair

AcDbSymbolTableIterator (Iterators)
   AcDbBlockTableIterator
   AcDbBlockTableRecordIterator
   AcDbDimStyleTableIterator
   AcDbLayerTableIterator
   AcDbLinetypeTableIterator
   AcDbRegAppTableIterator
   AcDbTextStyleTableIterator
   AcDbUCSTableIterator
   AcDbAbstractViewTableIterator
      AcDbViewTableIterator
      AcDbViewportTableIterator
   AcDbGroup Iterator

AcDbExtents

AcDbDate (Low-level datatypes)
AcDbIntArray
AcDbObjectIdArray
AcDbVoidPtrArray
```
10  Chapter 1  Overview

**AcRxObject**
AcDbDatabase (AutoCAD database)

AcDbDictionaryIterator
AcDbDwgFiler (Filers)
   AcDbDwgCopyFiler
   AcDbDeepCloneFiler
   AcDbWblockCloneFiler
   AcDbDwgUndoFiler
AcDbDxfFiler

(AcRxObject)
AcDbObject (The most important class)
AcDbDictionary

**AcDbEntity** (AutoCAD entities)
   AcDb3dSolid
   AcDbBlockBegin
   AcDbBlockEnd
   AcDbBlockReference
      AcDbMInsertBlock
   AcDbBody

**AcDbCurve**
   AcDb2dPolyline
   AcDb3dPolyline
   AcDbArc
   AcDbCircle
   AcDbEllipse
   AcDbLeader
   AcDbLine
   AcDbRay
   AcDbSpline
   AcDbXline

**AcDbDimension**
   AcDb2LineAngularDimension
   AcDb3PointAngularDimension
   AcDbAlignedDimension
   AcDbDiametricDimension
   AcDbOrdinateDimension
   AcDbRadialDimension
   AcDbRotatedDimension
   AcDbFace
   AcDbFaceRecord
   AcDbFcf
   AcDbMline
   AcDbMText
   AcDbOleFrame
   AcDbPoint
   AcDbPolyFaceMesh
   AcDbPolygonMesh
   AcDbRegion
   AcDbSequenceEnd
   AcDbShape
   AcDbSolid
   AcDbText
      AcDbAttribute
      AcDbAttributeDefinition
   AcDbTrace
AcDbVertex
  AcDb2dPolylineVertex
  AcDb3dPolylineVertex
  AcDbPolyFaceMeshVertex
  AcDbPolygonMeshVertex
  AcDbViewport
  AcDbZombieEntity

(AcRxObject)
(AcDbObject)
  AcDbGroup
  AcDbMlineStyle
  AcDbSymbolTable  (Symbol tables)
    AcDbBlockTable
    AcDbDimStyleTable
    AcDbLayerTable
    AcDbLinetypeTable
    AcDbRegAppTable
    AcDbTextStyleTable
    AcDbUCSTable
  AcDbAbstractViewTable
    AcDbViewportTable
    AcDbViewTable
  AcDbSymbolTableRecord  (Symbol table records)
    AcDbAbstractViewTableRecord
    AcDbViewportTableRecord
    AcDbViewTableRecord
    AcDbBlockTableRecord
    AcDbDimStyleTableRecord
    AcDbLayerTableRecord
    AcDbLinetypeTableRecord
    AcDbRegAppTableRecord
    AcDbTextStyleTableRecord
    AcDbUCSTableRecord
  AcDbZombieObject
  AcDbObjectIterator
  AcDbObjectReactor  (Used to mediate notification events)
    AcDbEntityReactor
    AcDbDatabaseReactor

**AcDb class hierarchy**

**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()`, which are part of the standard entity protocol. The `worldDraw()` function must be defined by all custom entity classes. The `AcGi World Draw` object provides an API through which `AcDbEntity::worldDraw()` can produce its graphical representation in all viewports simultaneously. Similarly, the `AcGi Viewport Draw` object provides an API through which `AcDbEntity::viewportDraw()` can produce different graphical representations for each viewport.
The class hierarchy for the AcGi library is shown in the following figure (abstract classes are shown in boldface). For more information on using AcGi classes, see chapter 10, “Deriving from AcDbEntity.”

\[
\begin{align*}
\text{AcRxObject} \\
\quad \text{AcGiEdgeData} \\
\quad \text{AcGiFaceData} \\
\quad \text{AcGiSubEntityTraits} \\
\quad \text{AcGiTextStyle} \\
\quad \text{AcGiVertexData} \\
\quad \text{AcGiViewport} \\
\quad \text{AcGiViewportDraw} \\
\quad \text{AcGiViewportGeometry} \\
\quad \text{AcGiWorldDraw} \\
\quad \text{AcGiWorldGeometry}
\end{align*}
\]

**AcGi class hierarchy**

**AcGe Library**

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

The class hierarchy for the AcGe library is shown in the following figures (abstract classes are shown in boldface). The library consists of two major subsets: classes for 2D geometry and classes for 3D geometry. The major abstract base classes are `AcGeEntity2d` and `AcGeEntity3d`. A number of basic classes that are 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 data members of these basic level classes are all declared public. These are the only classes in the geometry library that have public data members.

For more information on the different coordinate systems provided by the AcGe library, see appendix B, “Using the AcGe Library.” The sample programs in this manual illustrate numerous common uses of AcGe classes.

\[
\begin{align*}
\text{AcGePoint2d} \\
\text{AcGeVector2d} \\
\text{AcGeMatrix2d} \\
\text{AcGeScale2d} \\
\text{AcGePoint2dArray} \\
\text{AcGeVector2dArray}
\end{align*}
\]
AcGePoint3d
AcGeVector3d
AcGeMatrix3d
AcGeScale3d
AcGePoint3dArray
AcGeVector3dArray

AcGeTolerance
AcGeInterval
AcGeCurveBoundary
AcGeDoubleArray
AcGeKnotVector

AcGeEntity2d
  AcGePointEnt2d
    AcGePosition2d
    AcGePointOnCurve2d
  AcGeCurve2d
    AcGeLinearEnt2d
      AcGeLine2d
      AcGeRay2d
      AcGeLineSeg2d
      AcGeCircArc2d
      AcGeEllipArc2d
    AcGeSplineEnt2d
      AcGeCubicSpline2d
      AcGeNurbCurve2d
      AcGePolyLine2d
      AcGeExternalCurve2d

AcGeEntity3d
  AcGePointEnt3d
    AcGePosition3d
    AcGePointOnCurve3d
    AcGePointOnSurface
  AcGeCurve3d
    AcGeLinearEnt3d
      AcGeLine3d
      AcGeRay3d
      AcGeLineSeg3d
      AcGeCircArc3d
      AcGeEllipArc3d
      AcGeExternalCurve3d
    AcGeSplineEnt3d
      AcGeCubicSpline3d
      AcGeNurbCurve3d
      AcGePolyLine3d
      AcGeAugPolyLine3d
AutoLISP, ADS, and ARX

AutoLISP is an interpreted language that provides a simple mechanism for adding commands to AutoCAD. Although there is some variation depending on the platform, AutoLISP is logically a separate process that communicates with AutoCAD through interprocess communication (IPC), as shown in the following diagram.

ADS applications are written in C and are compiled. However, to AutoCAD, ADS applications appear identical to AutoLISP applications. An ADS application is written as a set of external functions that are loaded by and called from the AutoLISP interpreter. ADS applications communicate with AutoLISP by IPC.

The ARX programming environment differs from the ADS and AutoLISP programming environments in a number of ways. The most important difference is that an ARX application is a DLL that shares AutoCAD's address space and makes direct function calls to AutoCAD, avoiding the costly overhead of IPC. Applications that communicate frequently with AutoCAD run faster in the ARX environment than in the ADS or AutoLISP environments.
In addition to speed enhancements, you can add new classes to the ARX program environment and export them for use by other programs. ARX entities you create are virtually indistinguishable from built-in AutoCAD entities. You can also extend ARX protocol by adding functions at run time to existing AutoCAD classes.

Part of the ARX environment is a complete library of the ADS functions. This library, often referred to as ADS-Rx, is functionally identical to the standard C ADS library; however, it is actually implemented as a part of AutoCAD. Consequently, it shares AutoCAD's address space along with the other ARX libraries. Use the ADS library for the following:

- Entity selection
- Selection set manipulation
- Programmable dialog boxes
- AutoCAD utility requests, such as `ads_trans()`, `ads_command()`, and `ads_cmd()`
- Data acquisition

For more information on using ADS with ARX, see chapter 3, “Building an ARX Application,” and chapter 5, “Database Objects.”

**Registering Commands**

You can register new AutoCAD commands in both ADS (with the `ads_defun()` function) and in the ARX AcEd (with the `acedRegCmds()` macro). With the ADS library commands, requests are first routed to AutoLISP, then to the application. With ARX command registration, commands are added to AutoCAD’s built-in command set.

The way commands are registered affects how the commands can be invoked. For commands registered in ADS using `ads_defun()`

- The commands can be evaluated through AutoLISP or the `ads_invoke()` facility
- The commands cannot be invoked using the AutoLISP `command()` function or the `ads_command()` function
The opposite is true for commands registered through `acedRegCmds()`:

- The commands are not known to AutoLISP or the `ads_invoke()` facility.
- The commands can be invoked using the AutoLISP `command` function or the `ads_command()` function.

For more information on ADS functions, see the ADS Developer’s Guide. For more information on AutoLISP functions, see the AutoCAD Customization Guide.

**Entry Points**

ARX and ADS applications have different models for communicating with AutoCAD. An ADS application consists of a single, infinite loop that waits for AutoLISP requests. An ARX application has one main entry point that is used for messaging. Then, when you register commands, they become additional entry points into the application. When you override virtual functions for the C++ classes in the ARX libraries, those functions become entry points into the application as well.

**Comparing ADS and ARX Function Calls**

In general, the ARX API is much simpler than the ADS API. For example, in ARX you could use the following code to change the layer of a line:

```cpp
void changeLayer(const AcDbObjectId& entId,
                 const char* pNewLayerName)
{
    AcDbEntity *pEntity;
    acdbOpenObject(pEntity, entId, AcDb::kForWrite);
    pEntity->setLayer(pNewLayerName);
    pEntity->close();
}
```

Note: For simplicity, error checking is omitted from the chapter examples. Chapter 3 includes an example that illustrates use of appropriate error checking.

In ADS, the information about an entity is represented as a linked list of result buffers ("resbufs"). There are basically four steps required to change the layer of a line:

1. Use the `ads_entget()` function to obtain the entity information.
2. Look for the field that contains the layer value.
3. Change the field in the list.
4 Call `ads_entmod()` with the modified resbuf list to effect the change in the database.

The following is the C code for changing the layer of a line using ADS:

```c
void changeLayerADS(ads_name entityName,
                   const char* pNewLayerName)
{
    struct resbuf *pRb, *pTempRb;
    pRb = ads_entget(entityName);
    // No need to check for rb == NULL since all entities have a layer.
    for (pTempRb = pRb; pTempRb->restype != 8;
         pTempRb = pTempRb->rbnext) { ; }
    free(pTempRb->resval.rstring);
    pTempRb->resval.rstring = (char*) malloc(strlen(pNewLayerName) + 1);
    strcpy(pTempRb->resval.rstring, pNewLayerName);
    ads_entmod(pRb);
    ads_relrb(pRb);
    ads_retvoid();
}
```

Here is the AutoLISP code to do the same thing:

```lisp
(defun asdk_changeLayerLISP(ename newLayer / eList)
  (setq eList (entget ename))
  ; substitute the new layer name for the old
  (setq eList
    (subst (cons 8 newLayer) (assoc 8 eList) eList))
  ; Modify the entity's data in drawing to reflect the changed layer
  (entmod eList)
  (princ)
)
```

The following chart compares the AutoLISP, ADS, ADS-Rx, and ARX programming interfaces with respect to speed, exposure, power, and programming expertise required to use each API. The “exposure” parameter indicates the possible severity of your programming errors. Although the ARX interface is the most powerful of the four APIs compared here, it also offers the greatest potential for serious programming errors, such as corrupting AutoCAD data structures. The other programming environments require proportionately less programming expertise, but also provide less power and scope.
Chapter 1 Overview

Comparison of APIs

Run-Time Type Identification

Every subclass of \texttt{AcRxObject} can have an associated class descriptor object (of type \texttt{AcRxClass}) that is used for run-time type identification. ARX provides functions for testing whether an object is of a particular class or derived class, functions that allow you to determine whether two objects are of the same class, and functions for returning the class descriptor object for a given class. Important functions provided by the \texttt{AcRxObject} class for run-time type identification include the following:

- \texttt{desc()}, a static member function, returns the class descriptor object of a particular (known) class
- \texttt{cast()}, a static member function, returns an object of the specified type, or NULL if the object is not of the required class (or a derived class)
- \texttt{isKindOf()} returns whether an object belongs to the specified class (or a derived class)
- \texttt{isA()} returns the class descriptor object of an object whose class is unknown

When you want to know what class an object is, use \texttt{AcRxObject::isA()}. This function returns the class descriptor object (an instance of \texttt{AcRxClass}) for a database object. Its signature is

\begin{verbatim}
AcRxClass* isA() const;
\end{verbatim}
When you already know what class an object is, you can use the desc() function to obtain the class descriptor object:

```c
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.

```c
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.

```c
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()`.

```c
if (curEntity->isA() == AcDbEllipse::desc()) {
    // Got an AcDbEllipse, no more, no less.
}
The AutoCAD database stores the objects and entities that make up an AutoCAD drawing. This chapter introduces the key elements of the database: the named object dictionary and a set of symbol tables. Entities, the drawing objects that have a graphical representation, are owned by block table records that are stored in the block table, which is one of the database symbol tables.

Important concepts introduced in this chapter include object handles, object IDs, and the AcDb protocol for opening and closing objects. Sample code shows creating entities, layers, and groups and adding objects to the database.
AutoCAD Database

This section introduces key terms and concepts related to the AutoCAD database. The following section describes how different user commands affect this database. Sample code shows how to create and modify database objects using ARX.

An AutoCAD drawing is a collection of \texttt{AcDb} objects that are stored in a database. Each object in the database has a handle that is a unique identification for the object within the context of a particular drawing. Entities are a special kind of database object that have a graphical representation within an AutoCAD drawing, for example, lines, circles, arcs, text, solids, regions, splines, and ellipses. A user can see an entity on the screen and can manipulate it.

Other important database objects are symbol tables and dictionaries, objects that map a symbol name (a text string) to another \texttt{AcDb} object. AutoCAD provides a fixed set of symbol tables in the database, each of which contains instances of a particular class of symbol table record. Examples of symbol tables are the layer table (\texttt{AcDbLayerTable}), which contains layer table records, and the block table (\texttt{AcDbBlockTable}), which contains block table records. All AutoCAD entities are owned by block table records. The following figure shows these key components of the AutoCAD database.

Symbol tables and dictionaries are both containers used to store objects. Dictionaries were introduced in AutoCAD Release 13 and provide a more generic container for storing objects. A dictionary can contain any \texttt{AcDbObject} or subclass thereof. You can create new database elements and add them to a new dictionary, also created by you. You cannot add a new symbol table to the database.

During AutoCAD edit sessions, you can obtain the database for the current drawing by calling the global function \texttt{acdbCurDwg()}. 
Multiple Databases

Multiple databases can be loaded within a single AutoCAD session. Each object in the session has an object ID, which is a unique identifier for the object across all databases loaded at one time. In contrast to the object ID, an object handle is not guaranteed to be unique within an AutoCAD session. It is unique only within the scope of a particular AcDbDatabase.

Note Although ARX allows you to load multiple databases, the AutoCAD editor will work with only one drawing at a time, and the editor must load the drawing from the disk file.

Object ID

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 Objects” on page 28.

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.

■ Use database protocol for obtaining the object ID of the objects that are created automatically whenever 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 object IDs of the selected objects. (For more information on selection sets, see chapter 6, “Entities.”)
Essential Database Objects

As objects are created, 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). The scenario in “Creating Objects” and “Sample ARX Code” details this process.

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

- A set of nine symbol tables, which includes the block table and the layer table. The block table already contains two records: *MODEL_SPACE and *PAPER_SPACE. The layer table already contains one record, layer 0.
- 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.

These objects can be automatically created in a new database object by passing kTrue in for its constructor’s buildDefaultDrawing argument. Passing in kFalse prepares the database object for loading in a DWG or DXF file.

Creating Objects

This section describes creating a line, a circle, and a layer and adding them to the database. First, suppose the user creates a line in model space with the following command:

```
line 3,2 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 basic 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 cause the entity to be added to the current database as well as the model space block. You can ask any entity what database it belongs to, and what 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:

\textit{group 3, 2, 9, 3}

Behind the scenes, AutoCAD creates a new group and adds it to the group dictionary, which is contained in the named object dictionary.

**Sample ARX Code**

The sample 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 is also shown.

**Creating Entities**

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

```cpp
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;
 acdbCurDwg() -> getBlockTable(pBlockTable,
 AcDb::kForRead);

 AcDbBlockTableRecord *pBlockTableRecord;
 pBlockTable -> getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
 AcDb::kForWrite);
```
pBlockTable->close();
AcDbObjectId lineld;
pBlockTableRecord->appendAcDbEntity(lineld, pLine);
pBlockTableRecord->close();
pLine->close();
return lineld;
}

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.

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

```cpp
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;
    acdbCurDwg()->getBlockTable(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.

```cpp
void
createNewLayer()
{
    AcDbLayerTable *pLayerTable;
    acdbCurDwg()->GetLayerTable(pLayerTable,
        AcDb::kForWrite);

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

// The linetype objectId default is 0 which is
// not a valid Id. So, it must be set to a
// valid Id...say for the CONTINUOUS linetype.
// Other data members have valid defaults so
// they can be left alone.
//
AcDbLinetypeTable *pLinetypeTbl;
acdbCurDwg()->getLinetypeTable(pLinetypeTbl,
    AcDb::kForRead);

AcDbObjectId ltypeObjId;
pLinetypeTbl->getAt("CONTINUOUS", ltypeObjId);
pLayerTableRecord->setLinetypeObjectId(ltypeObjId);
pLayerTable->add(pLayerTableRecord);
pLayerTable->close();
pLayerTableRecord->close();
}

Opening and Closing 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 ARX 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 the following example shows:

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:

pObject->close();

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

Acad::ErrorStatus
colorEntity(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. Some functions, such as AcDbBlockTable::getAt(), 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. In addition, you own the object and can freely delete it at any time before the object is added to the database. After that, the database owns it, and deleting the object 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 createLine() and createCircle() and puts the group into the group dictionary. Notice how the group dictionary is opened for writing, modified, and then explicitly closed.

```cpp
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, which resides in the named object dictionary.
    AcDbDictionary *pGroupDict;
    acdbCurDwg()->getGroupDictionary(pGroupDict, AcDb::kForWrite);
    AcDbObjectId pGroupId;
    pGroupDict->setAt(pGroupName, pGroup, pGroupId);
    pGroupDict->close();
    pGroup->close();
}
```
This chapter describes how to write and run an ARX application. It lists the messages passed by AutoCAD to the ARX application and shows how the application typically responds to those messages. Registering new commands with AutoCAD is also described.
ARX Application Structure

ARX applications are dynamic linked libraries (DLLs). AutoCAD calls into the ARX module through `acrxEntryPoint()`, which replaces the main of a C or ADS program. You are responsible for implementing the `acrxEntryPoint()` function, as described in this section.

The `acrxEntryPoint()` function serves not only as the entry point for communicating with an ARX application from AutoCAD, but also as a way for ARX applications to pass messages to the application and to return status codes to AutoCAD. All requests to invoke functions through `ads_defun()` are made by `acrxEntryPoint()`. If you define a new command with ARX or with `ads_regfunc()`, AutoCAD immediately executes the function associated with the command (see "Example Application" on page 40).

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

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

where

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

Within the definition of `acrxEntryPoint()`, you write a switch statement to decipher messages from AutoCAD, to perform appropriate actions related to the message, and to return an integer status value. The following code shows the skeleton of such a switch statement.

```cpp
AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
    switch(msg) {
    case AcRx::kInitAppMsg:
        break;
    ...
    default:
        break;
    } return AcRx::kRetOK;
}
```
### AutoCAD Messages to ARX Applications

The following table describes the messages that AutoCAD sends to ARX applications.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kInitAppMsg</td>
<td>Sent when the application is loaded, to open communications between AutoCAD and the application</td>
</tr>
<tr>
<td>kUnloadAppMsg</td>
<td>Sent when the ARX application is unloaded (either when the user unloads the application or when AutoCAD itself is terminated); closes files, performs clean-up operations</td>
</tr>
<tr>
<td>kLoadDwgMsg</td>
<td>Sent when the drawing is opened; makes the ADS library available to the application functions that need it</td>
</tr>
<tr>
<td>kUnloadDwgMsg</td>
<td>Sent when the user quits a drawing session; unloads the ADS library from memory</td>
</tr>
<tr>
<td>kInvkSubrMsg</td>
<td>Sent to invoke functions registered using ads_defun()</td>
</tr>
<tr>
<td>kEndMsg</td>
<td>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.</td>
</tr>
<tr>
<td>kQuitMsg</td>
<td>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, the kQuitMsg is sent.</td>
</tr>
<tr>
<td>kSaveMsg</td>
<td>Sent when AutoCAD is saving the drawing because a SAVE, SAVEAS, NEW, or OPEN command is entered</td>
</tr>
<tr>
<td>kCfgMsg</td>
<td>Sent when AutoCAD returns from the configuration program; implemented only for a change to the display driver</td>
</tr>
<tr>
<td>kNullMsg</td>
<td>An empty or zero message</td>
</tr>
</tbody>
</table>
Chapter 3 Writing an ARX Application

If an application contains only ARX-defined commands and there is no interaction with AutoLISP, the only messages you will receive are `kInitAppMsg`, `kLoadDwgMsg`, and `kUnloadAppMsg`. You won’t receive `kSaveMsg`.

You cannot use an `ads_command()` function for `kLoadDwgMsg`. Also, an ARX application cannot invoke an old-style ADS application that makes a call to `ads_command()`.

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

Sequence of Events in an ARX Application

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Message Sent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start AutoCAD</td>
<td><code>kInitAppMsg</code></td>
</tr>
<tr>
<td></td>
<td>Open drawing 1</td>
<td><code>kLoadDwgMsg</code></td>
</tr>
<tr>
<td></td>
<td>Invoke TEST1 command</td>
<td><code>ads_defun &quot;c: TEST1&quot;</code></td>
</tr>
<tr>
<td></td>
<td>Invoke TEST2 command</td>
<td><code>kInvkSubr</code></td>
</tr>
<tr>
<td></td>
<td>Invoke SAVE command</td>
<td>Control transfers directly to routine “test2”</td>
</tr>
<tr>
<td></td>
<td>Open drawing 2</td>
<td><code>kUnloadDwgMsg</code></td>
</tr>
<tr>
<td></td>
<td>Quit</td>
<td><code>kQuit</code></td>
</tr>
</tbody>
</table>

If an application is loaded when a drawing is already open, the `kInitAppMsg` and `kLoadDwgMsg` messages are sent in succession. When an ARX application is unloaded while an edit session is in progress, the `kUnloadDwg` and `kUnloadApp` messages are sent in succession.
The following table describes recommended action upon receipt of a given message.

<table>
<thead>
<tr>
<th>Message</th>
<th>Recommended Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>kInitAppMsg</td>
<td>Do register services, classes, AcEd commands and reactors, and AcRxDynamicLinker reactors. Initialize application's system resources, such as devices and windows. Perform all one-shot 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/ADS to be present, or any databases to be open. Calls involving any of these assumptions will result in an error condition, sometimes fatal. AcDb, AcGi, and ADS libraries are generally not yet active, although related AcRx and other structures are in place.</td>
</tr>
<tr>
<td>kUnloadAppMsg</td>
<td>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.</td>
</tr>
<tr>
<td>kLoadDwgMsg</td>
<td>Do perform initialization relevant to the current drawing edit session. AcDb, AcGi, ADS, 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/ADS function registration at this time, as well as 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 acdbCurDwg(). 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.</td>
</tr>
</tbody>
</table>
Chapter 3 Writing an ARX Application

Registering New Commands

This section describes adding new commands using the AcEd command registration mechanism. For information on adding new commands using the ADS functions `ads_defun()` and `ads_regfunc()`, see the ADS Developer's Guide. For information on adding new commands using the AutoLISP `defun` function, see the AutoCAD Customization Guide.

<table>
<thead>
<tr>
<th>Message</th>
<th>Recommended Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>kUnloadDwgMsg</td>
<td>Do release or deregister 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. Also, don’t deregister AcRx classes, AcEd reactors, or commands, because they remain valid across edit sessions.</td>
</tr>
<tr>
<td>kInvkSubrMsg</td>
<td>Do invoke the functions registered with <code>ads_defun()</code>. Determine the function by making a call to <code>ads_funcode()</code>. Do return values with <code>ads_retxxx()</code>. Don’t do much here except function invocation.</td>
</tr>
<tr>
<td>kEndMsg</td>
<td>Do the same things you would have done for the similarly named return codes from <code>ads_link(*)</code>: RQEND, RQCFG, RQQUIT, RQSAVE. See the ADS Developer’s Guide for more information on these return codes, which applies equally to these ARX messages. 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.</td>
</tr>
</tbody>
</table>

---

Registering New Commands

This section describes adding new commands using the AcEd command registration mechanism. For information on adding new commands using the ADS functions `ads_defun()` and `ads_regfunc()`, see the ADS Developer’s Guide. For information on adding new commands using the AutoLISP `defun` function, see the AutoCAD Customization Guide.
Registering New Commands

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 native AutoCAD commands, as well as any custom commands you add to it. The `acedRegCommands()` 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.

You usually add commands one at a time with `AcEdCommandStack::addCommand()`, and you remove commands by group with `AcEdCommandStack::removeGroup()`. 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 `addCommand()` is

```cpp
Acad::ErrorStatus AcEdCommandStack::addCommand
    (const char* cmdGroupName,
     const char* cmdGlobalName,
     const char* cmdLocalName,
     Adesk::Int32 commandFlags,
     AcRxFunctionPtr functionAddr);
```

where

- `cmdGroupName` ASCII representation of the group to add the command to. If the group doesn’t exist, then 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 38).

- `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` (see “Transparent Commands” on page 39), `ACRX_CMD_USEPICKSET`, and `ACRX_CMD_REDRAW`.

- `functionAddr` Address of the function to be executed when this command is invoked by AutoCAD.
The signature for `removeCmd()` is

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

The signature for `removeGroup()` is

```cpp
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 be searched first. At the user level, the ARX command Group option 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 global name can be used for both names.

To invoke a command using the AutoLISP `command` function or the `ads_command()` function, use the global name prefixed by an underscore (_) character.
The following code overrides the `AcEditorReactor::commandEnded()` function to determine whether the user entered a global or local command name. (For more information on the editor reactor, see chapter 11, “Notification.”)

```cpp
void ASDKCommandReactor::commandEnded(const char *pCmdStr)
{
    // Print the user typed name. Get both the global name
    // and local name and print that out too.
    ads_printf("\nUser entered : %s", pCmdStr);

    AcEdCommand* pCmd = NULL;
    if ((pCmd = acedRegCmds->lookupCmd(pCmdStr,
        Adesk::kTrue, Adesk::kTrue)) != NULL)
    {
        ads_printf("\nGlobal name : %s",
            pCmd->globalName());
        ads_printf("\nLocal  name : %s",
            pCmd->localName());
    } else if ((pCmd = acedRegCmds->lookupCmd(pCmdStr,
        Adesk::kFalse, Adesk::kTrue)) != NULL)
    {
        ads_printf("\nGlobal name : %s",
            pCmd->globalName());
        ads_printf("\nLocal  name : %s",
            pCmd->localName());
    } else {
        ads_printf("\nThis is not an ARX-registered'
" command");
    }
}
```

**Transparent Commands**

A command can be either modal or transparent. 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 `addCommand()` function specifies whether the new command is modal (ACRX_CMD_MODAL) or transparent (ACRX_CMD_TRANSPARENT).

Transparent commands currently can be nested only one level (that is, the main command is invoked, which invokes one transparent command). To invoke a transparent command using the AutoLISP `command` function or the `ads_command()` function, prefix the command name with an apostrophe (').

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.
Querying Commands

The AcEd library offers classes for iterating through the command stack and for looking up specific commands. However, this querying is usually performed by AutoCAD, and you will rarely use these functions.

The `AcEdCommandIterator` class contains the `nextCommand()` and `curCommand()` member functions for stepping through the command stack. For more information on these functions, see the ARX Reference Manual.

The `AcEdCommandStack::lookupCmd()` function allows you to look up a command by name. This function is usually called by AutoCAD rather than by an application.

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`, which is described in chapter 8, "Defining a Custom ARX Class") and adds it to the ARX hierarchy with the `acrxBuildClassHierarchy()` function.

```cpp
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();
}
```

```cpp
// Clean up function called from acrxEntryPoint during the // kUnloadAppMsg case. This function removes this app's // command set from the command stack and removes this app's // custom classes from the ACRX run-time class hierarchy.

void unloadApp()
{
    acedRegCmds->removeGroup("ASDK_DICTIONARY_COMMANDS");

    // Remove the AsdkMyClass class from the ACRX runtime // class hierarchy. If this is done while database is // still active, it should cause all objects of class // AsdkMyClass to become zombies.
    //
    deleteAcRxClass(AsdkMyClass::desc());
}
```
Loading an ARX Application

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

- Specify the application in the initial module file, acad.rx. This file contains ASCII text with the names of all programs that are to be autoloaded when AutoCAD 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, $ACAD). The acad.rx file must also be in a directory on the AutoCAD search path.

- Make an application load request from another ARX application using the C function acrxLoadModule() or the C++ function AcRxDynamicLinker::loadModule().

- Use the APPLOAD dialog defined in the AutoCAD support file appload.lsp. This file defines a user interface for the AutoLISP arxload and arxunload functions.

- From AutoLISP, use the arxload function.

- From ADS or from ARX, use the ads_arxload() function. For an ARX application, the difference between this function call and acrxLoadModule() is that this call causes the application to be registered as an ADS application as well as an ARX application.

- Enter the ARX command on the AutoCAD command line (see “ARX Command” on page 43).

Unloading an ARX Application

You can unload an ARX application with any of the following methods:

- Make an application unload request from another ARX application using the C function acrxUnloadModule() or the C++ function AcRxDynamicLinker::unloadModule().

- Use the APPLOAD dialog defined in the AutoCAD support file appload.lsp. This file defines a user interface for the AutoLISP arxload and arxunload functions.

- From AutoLISP, use the arxunload function (uses $ACAD path).

- From ADS or from ARX, use the ads_arxunload() function (uses $acad path).

- Enter the ARX command on the AutoCAD command line and use the Unload option (uses $acad path).
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.

It is up to the application to perform the appropriate bookkeeping for unlocking. An application can take one of two basic approaches. It can keep track of all pointers it passes to other applications or to AutoCAD and allow itself to be unloaded only when there are no active client pointers to its elements. Or, it can actively force unloading by keeping track of all pointers, and then notifying clients to forget all such active pointers when the application is unloaded.

Examples of cleanup operations an application must perform in order to be unloadable include the following:

- Removing any new classes it created using the deleteAcRxClass() function. Classes must be removed starting with the most derived classes first, working up the class tree to parent classes.
- Deleting 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 AcDbZombieObject or AcDbZombieEntity.
- Removing reactors that have been attached to any AcDbObject, AcDbDatabase, AcRxDynamicLinker, or AcEditor object. (Persistent reactors on AcDbObjects are an exception: they will become zombies when the application is unloaded.)
- Removing any commands registered with acedRegCmds() or ads_defun().

If you want to make your application unloadable, you need to store the value of the pkt parameter in acrxE EntryPoint() sent with the AcRx::kInitAppMsg. 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:

```cpp
Adesk::Boolean AcRxDynamicLinker::lockApplication(void* pkt) const;
Adesk::Boolean AcRxDynamicLinker::unlockApplication(void* pkt) const;
```
The following function checks whether a given application is unlocked:

```cpp
Adesk::Boolean AcRxDynamicLinker::isApplicationLocked(const char* name) const;
```

Analogous functions are also provided in C:

```c
int acrxLockApplication(void* pkt);
int acrxUnlockApplication(void* pkt);
int acrxApplicationIsLocked(const char* modulename);
```

ARX Command

The following sections describe the ARX command and its options. The initial prompt is shown in the following:

```
?/Load/Unload/Commands/Options: Enter an option or press Return
```

?—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 ARX Application” on page 41 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 they had 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's 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 `ads_defun()` by ADS and ARX applications. The software mechanism that defines command groups is described in “Lookup Order” on page 38.
Error Handling

- **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.

### 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”.

```cpp
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;
  Acad::ErrorStatus es = acdbCurDwg()->getBlockTable(
    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("App 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("App X failed to close"n
                " Model Space Block Record. Error: %s",
                acadErrorStatusText(es2));
    }
    delete pCirc;
    return es;
}

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

es = pCirc->close();
if (es != Acad::eOk) {
    acrx_abort("App 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 = acdbCurDwg()->getLayerTable(pLayerTable,
            AcDb::kForWrite);
    if (es != Acad::eOk) {
        delete pLayerTableRecord;
        return es;
    }

}
// The linetype objectId default is 0 which is not a valid Id. So, it must be set to a valid Id...for example the CONTINUOUS linetype. Other data members have valid defaults so they can be left alone.

AcDbLinetypeTable *pLinetypeTbl;
es = acdbCurDwg()->getLinetypeTable(pLinetypeTbl, AcDb::kForRead);
if (es != Acad::eOk) {
    delete pLayerTableRecord;
es = pLayerTable->close();
    if (es != Acad::eOk) {
        acrx_abort("App 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("App 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("App X failed to close Layer Table. Error: %d", acadErrorStatusText(es2));
                delete pLayerTableRecord;
                return es;
            }
            es = pLayerTable->close();
            if (es != Acad::eOk) {
                acrx_abort("App X failed to close Layer Table. Error: %d", acadErrorStatusText(es));
                return es;
            }
        }
    }
}
    return es;
}
es = pLayerTableRecord->close();
if (es != Acad::eOk) {
    acrx_abort("An App X failed to close Layer Table Record. Error: %d",
               acadErrorStatusText(es));
    return es;
}

Memory Management

Objects must be freed by the memory manager that allocated them. For example, in C use malloc or calloc to allocate memory and free to release it. In C++, use AcDbObject::new to create an object and AcDbObject::delete to destroy it (unless it has been added to the database).

Note Both malloc and free are mapped to ads_malloc() and ads_free() by means of #defines, which causes the application to use AutoCAD’s memory manager.

It is possible to suppress or override the #defines and use a separate memory manager. (Use ads_malloc() and ads_free() for creating and freeing buffers that will be used by the ADS library.) However, all contracts involving malloc, free, new, and delete for interacting with AutoCAD must be adhered to. For example, a resbuf chain passed in to ads relrb() should have been created either by a call to ADS or through a direct call to ads malloc(). It cannot have been allocated through any other memory manager. Also, any object created by AcDb or ADS or any other ARX library must be deleted or freed by the corresponding function. Do not attempt to allocate multiple objects in a single array.
This chapter describes basic database protocol, including how to create a database, read in a drawing file, and save the database. The `wblock` and `insert` operations are also described here.

For a more detailed discussion of the `deepClone` and `wblock` operations, see chapter 13, “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 already contains two records: `*MODEL_SPACE` and `*PAPER_SPACE`. 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 it). When first created, the database is empty. To populate it, you either read in a drawing file, or use the `AcDbDatabase::buildDefaultDrawing()` function to create the standard database objects, described in the previous section, "Initial Database" on page 50.

Use the following function to read in a drawing file:

```cpp
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

The `AcDbDatabase::closeInput()` function forces the drawing load to be completed so that the underlying file is released and other applications can access it. This function is not generally used unless you have a special need to free the drawing for use by others.

Never delete the database pointed to by `acdbCurDwg()`.

### Saving a Database

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

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

### 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:

```cpp
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:

```cpp
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:

```cpp
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 is invoked on. AutoCAD merges the objects that it defines, such as the MLINESTYLE 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 13, “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:

```cpp
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:

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

#### Color

If a color is not explicitly 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:

```cpp
Acad::ErrorStatus
AcDbDatabase::setColor(const AcCcColor& color);
```

```cpp
AcCcColor AcDbDatabase::setColor();
```
Chapter 4 Database Operations

**Linetype**

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

```cpp
Acad::ErrorStatus AcDbDatabase::setCeltype(AcDbObjectId);
AcDbObjectId AcDbDatabase::celtype() const;
```

**Linetype Scale**

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 (for more information, see chapter 6, “Entities”). Use the following functions to set and inquire these values:

```cpp
Acad::ErrorStatus AcDbDatabase::setLtscale(double);
double AcDbDatabase::ltScale() const;
```

```cpp
Acad::ErrorStatus AcDbDatabase::setCeltscale(double);
double AcDbDatabase::celtscale() const;
```

```cpp
Acad::ErrorStatus AcDbDatabase::setPsltscale(Adesk::Boolean)
Adesk::Boolean AcDbDatabase::psltscale() const;
```
Layer

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

```cpp
Acad::ErrorStatus
AcDbDatabase::setClayer(AcDbObjectId);
AcDbObjectId clayer() const;
```

External References

External references (xrefs) are currently created only by AutoCAD; they cannot be created through the AcDb interface. For information on how to create and use xrefs, see the Release 13 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 11, “Notification,” to monitor xref events to copy in or record the location of application-defined objects whose ownership is rooted in the named object dictionaries of the xref databases. The `AcEditorReactor` class provides the following reactor callback functions:

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

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 ARX Reference Manual.

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

```cpp
void createDwg()
{
    AcDbDatabase *pDb = new AcDbDatabase;
    AcDbBlockTable *pBtbl;
    pDb->getBlockTable(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);
    *pCir1 = 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()
{
    AcDbDatabase *pDb = new AcDbDatabase;

    // AcDbDatabase::readDwgFile() function
    // automatically appends a DWG extension if it's not
    // specified in the filename parameter.
    // pDb->readDwgFile("test1.dwg");
}
```
Example of Database Operations

```c++
// open the model space block table record
AcDbBlockTable *pBlkTbl;
pDb->getBlockTable(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);
    ads_printf("classname: %s
",
          (pEnt->isA())->name());
    pEnt->close();
}
pBlkTblRcd->close();
delete pBlkTblRcdItr;
delete pDb;
```
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 Objects

Each AcDbObject 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. 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. This pointer is valid until the object is closed, as shown in the following figure.

The function for opening an object is `AcdbOpenObject()`:

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

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

```cpp
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:

```cpp
AcDbObject* pObject;
AcDbHandle handle;

pObject->getAcDbHandle(handle);
```

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`:

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

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

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:

```cpp
AcDbEntity*
selectEntity(AcDbObjectId& eId, AcDb::OpenMode openMode)
{
    ads_name en;
    ads_point pt;
    ads_entsel("Select 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 11, “Notification,” for more information. Applications will rarely need to open an object for notify and send it notification.

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

### Opening objects in different modes

<table>
<thead>
<tr>
<th>Request for- &gt;</th>
<th>kForRead</th>
<th>kForWrite</th>
<th>kForNotify</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectAlready Opened for:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>openedForRead</td>
<td>eAtMaxReaders if readCount = 256; otherwise succeeds</td>
<td>eWasOpenForRead</td>
<td>succeeds</td>
</tr>
<tr>
<td>openedForWrite</td>
<td>eWasOpenForWrite</td>
<td>eWasOpenForWrite</td>
<td>succeeds</td>
</tr>
<tr>
<td>openedForNotify</td>
<td>eWasOpenForNotify</td>
<td>eWasOpenForNotify</td>
<td>eWasOpenForNotify</td>
</tr>
<tr>
<td>wasNotifying</td>
<td>eWasNotifying</td>
<td>eWasNotifying</td>
<td>eWasNotifying</td>
</tr>
<tr>
<td>Undo</td>
<td>eWasOpenForUndo</td>
<td>eWasOpenForUndo</td>
<td>succeeds</td>
</tr>
</tbody>
</table>

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 read 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 283.
Deleting Objects

When you create an instance of an `AcDbObject` with the intent of appending it to the database, use `AcDbObject::new`. 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.

Ownership

An object that is implicitly owned by the database rather than another database object is called a root object. The database contains 10 root objects: the 9 symbol tables and the named object dictionary. All filing operations begin by filing out the root objects of the database. For more information, see “Object Filing” on page 72.

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:

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

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

AutoCAD ownership connections are as follows:

- Block table records own entities
- Each symbol table owns a particular type of symbol table record
- An `AcDbDictionary` can own any `AcDbObject`
- Any `AcDbObject` 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 dictionary of any object
- Custom objects that can hold data (see chapter 9, “Deriving from AcDbObject”)

Extended Data

Extended data (xdata) is created by applications written with ADS 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:

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

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

```cpp
virtual Acad::ErrorStatus AcDbObject::setXData(const resbuf* xdata);
```
The following example uses xData() 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 setXdata() to modify the object’s xdata. This example also illustrates the use of upgradeOpen() and downgradeOpen().

// This is the main function of this app. It 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 to have the
// restype and resval values listed out.
//
void
printXdata()
{
    // Select and open an object.
    //
    AcDbObject *pObj;
    if ((pObj = selectObject(AcDb::kForRead)) == NULL) {
        return;
    }

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

    // Get the xdata for appname.
    //
    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
        // ADS and LISP because this is ONLY the Xdata so
        // the -3 Xdata-start marker isn’t needed.
        //
        printList(pRb);
        ads_relrb(pRb);
    }
}
void addXdata()
{
    AcDbObject* pObj = selectObject(AcDb::kForRead);
    if (!pObj) {
        ads_printf("Error selecting object\n");
        return;
    }

    // Get the application name and string to be added to
    // xdata
    char appName[100], resString[200];
    appName[0] = resString[0] = '\0';
    ads_getstring(NULL, "Enter application name: ", appName);
    ads_getstring(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
        // ADS and LISP because this is ONLY the Xdata so
        // the -3 Xdata-start marker isn’t needed.
        //
        ads_regapp(appName);

        pRb = ads_newrb(AcDb::kDxfRegAppName);
        pTemp = pRb;
        pTemp->resval.rstring = (char*) malloc(strlen(appName));
        strcpy(pTemp->resval.rstring, appName);
    }
}
Adding Object-Specific Data

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

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

ARX Example

The following example shows instantiating an xrecord and adding it to a 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();
}
Chapter 5  Database Objects

// Get the objectId 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 = ads_buildlist(AcDb::kDxfText,
"This is a test Xrecord list",
AcDb::kDxfCoord, 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(*head);
pXrec->close();
ads_relrb(head);
}

// 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 objectId of the object's extension dictionary

// dictObjId = pObj->extensionDictionary();
pObj->close();

// Get the objectId 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 = ads_buildlist(AcDb::kDxfText,
"This is a test Xrecord list",
AcDb::kDxfCoord, 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(*head);
pXrec->close();
ads_relrb(head);
}
Adding Object-Specific Data

ADS Example

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

```c
int createXrecord()
{
    ads_point dummy, testpt = {1.0, 2.0, 0.0};
    ads_name xrecname, ename, extDict = {0L, 0L};

    // Have user select an entity. Then get it's data.
    //
    if (ads_entsel("select entity: ", ename, dummy) != RTNORM)
    {
        ads_printf("Nothing selected");
        ads_retvoid();
        return RTNORM;
    }
    pEnt = ads_entget(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.rname, ename, extDict);
                break;
            }
        }
    }
}
```
// If no extension dictionary exists add one
//
if (extDict[0] == 0L) {
    pDict = ads_buildlist(RTDXF0, "DICTIONARY", 100,
        "AcDbDictionary", 0);
    ads_entmakex(pDict, extDict);
    ads_relrb(pDict);
    pDict = ads_buildlist(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;
    ads_entmod(pEnt);
    ads_relrb(pEnt);
}

// At this point the entity has an extension dictionary.
// So, create a resbuf list of the xrecord's entity info
// and data.
//
pXrec = ads_buildlist(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.
//
ads_entmakex (pXrec, xrecname);
ads_relrb(pXrec);

// Set the Xrecord's owner to the extension dictionary
//
ads_dictadd(extDict, 'ASDK_XRECADS', xrecname);
ads_retvoid();
return RTNORM;
Adding Object-Specific Data

// Accesses the xrecord associated with the key "ASDK_XRECADS" in
// the extension dictionary of a user selected entity. It then
// lists 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 user select an entity. Then get it's data
  if (ads_entsel("select entity: ", ename, dummy) != RTNORM) {
    ads_printf("Nothing selected");
    ads_retvoid();
    return RTNORM;
  }
  pEnt = ads_entget(ename);

  // Now 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.rname,
                     extDict);
        break;
      }
  }

  if (extDict[0] == 0L) {
    ads_printf("No extension dictionary present.");
    return RTNORM;
  }

  pXrec = ads_dictsearch(extDict, "ASDK_XRECADS", 0);
  if (pXrec) {
    printList(pXrec);
    ads_relrb(pXrec);
  }
  ads_retvoid();
  return RTNORM;
}
Erasing Objects

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

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

AutoCAD marks the object as erased, but the object remains in the database. You can unerase an object by setting the `Erasing` parameter to `FALSE`. Erased objects are not filed out to DWG or DXF files.

You cannot open an erased object unless you specifically ask for it. Otherwise, the `eWasErased` error code is returned when you try to open an erased object. The last parameter of the following function allows you to specify that you want to open an erased object. (`kFalse`, the default, specifies not to open erased objects.)

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

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

Object Filing

Object filing refers to the process of converting 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 between AutoCAD, AutoLISP, ADS, and ARX (uses DXF format)
- Undo recording and restoring (uses DWG format)
- Various 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. For more detailed information, see chapter 9, “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, 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 9, “Deriving from AcDbObject”).
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.
An entity is a database object that has a graphical representation. Examples of entities include lines, circles, arcs, text, solids, regions, splines, and ellipses. A user can see an entity on the screen and can manipulate it. The figure on the following page shows the class hierarchy for AcDbEntity, which 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 are 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 the section, “Complex Entities” on page 111.

Ownership

Entities in the database normally belong to an AcDbBlockTableRecord. The block table in a newly created database has two predefined records, *MODEL_SPACE and *PAPER_SPACE. Additional records are added whenever the user creates new blocks (block records), typically by issuing a BLOCK, HATCH, or DIMENSION command.

The following figure shows the ownership structure for database entities (abstract classes are shown in boldface and complex entities are marked with the † symbol).
Ownership structure for database entities
AutoCAD Release 12 Entities

The following entities were included in AutoCAD Release 12 and are declared in the dbents.h file:

Note You cannot currently derive new classes from Release 12 entities. Most, but not all, of their member functions have been implemented in AutoCAD Release 13.

- AcDbText
- AcDbAttribute
- AcDbAttributeDefinition
- AcDbBlockBegin
- AcDbBlockEnd
- AcDbSequenceEnd
- AcDbBlockReference
- AcDbMInsertBlock
- AcDb2dPolylineVertex
- AcDb3dPolylineVertex
- AcDbPolygonMeshVertex
- AcDbPolyFaceMeshVertex
- AcDbFaceRecord
- AcDb2dPolyline
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

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 methods for setting and getting the current property values associated with the database.
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 member function `AcCmColor::getColorIndex()`.

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

### Colors 1 to 7

<table>
<thead>
<tr>
<th>Color number</th>
<th>Color name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>2</td>
<td>Yellow</td>
</tr>
<tr>
<td>3</td>
<td>Green</td>
</tr>
<tr>
<td>4</td>
<td>Cyan</td>
</tr>
<tr>
<td>5</td>
<td>Blue</td>
</tr>
<tr>
<td>6</td>
<td>Magenta</td>
</tr>
<tr>
<td>7</td>
<td>White or Black</td>
</tr>
</tbody>
</table>

Colors 8 through 255 are defined by the display device.

The following index values have special meanings:

0 Specifications 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 Specifications BYLAYER. Entities assume the color of the entity's associated layer (an `AcDbLayerTableRecord`).

257 No color. Only present from the time an entity is first instantiated until either its color is set to a value between 0 and 256, or when 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:

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

### 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** The default linetype, which is automatically created in the linetype symbol table.
- **BYLAYER** Specifies to use the linetype value of the entity's layer.
- **BYBLOCK** Specifies to use the 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 allow you to set the linetype for an entity, either by name or by object ID:

```cpp
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:

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

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

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

```cpp
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 non-paper 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.

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 OFF or frozen. The value of `AcDb::Visibility` can be either `kInvisible` or `kVisible`. 
Common Entity Functions

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 allow you to set the layer for an entity, either by name or by object ID:

\[
\text{Acad::ErrorStatus Acad::Entity::setLayer(const char* newVal);} \\
\text{Acad::ErrorStatus Acad::Entity::setLayer(AcDbObjectId newVal);} \\
\text{This function returns the name of the current entity layer:} \\
\text{char* Acad::Entity::layer}();\text{ const;} \\
\text{This function returns the object ID for the current layer (an object of type AcDbLayerTableRecord):} \\
\text{AcDbObjectId Acad::Entity::layerId}();\text{ const;}
\]

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 10, “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 is usually the same.
- `moveStretchPointsAt()` is used by the AutoCAD STRETCH command to move specified points. Defaults to `transformBy()`.
- `moveGripPointsAt()` is used by AutoCAD grip editing to move specified points. 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; produces `ads_printf()` 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 87).
- `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 87).

**Note** The following functions are not currently implemented for AutoCAD Release 12 entities: `getTransformedCopy()`, `worldDraw()`, `viewportDraw()`, and `list()`.
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. Possible Object Snap modes are:

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

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

```cpp
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:

```cpp
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 100).
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 **kExtendArg** 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 `intersectWith()`:

![Diagram of lines](image_url)
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 intersectWith() allow you to supply optional GS markers to optimize performance for this function. Supplying GS markers for the entities to be tested localizes the intersection area and speeds 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, which are contained in the AcGi library. Any class derived from AcDbEntity can associate a GS (graphics system) 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 \texttt{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.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{box.png}
\caption{Diagram of a box with GS markers labeled 1 to 12.}
\end{figure}

An entity is composed of subentities, which can be of the following type: vertex, edge, or face. Currently, the only entities that support subentities are bodies, regions, solids, and mlines. Use the \texttt{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 \texttt{AcDbFullSubentPath}, consists of an array of object IDs and a subentity ID object:

\begin{verbatim}
{AcDbObjectIdArray mObjectIds;
 AcDbSubentId mSubentId; }
\end{verbatim}

The array contains the object IDs that specify the path to the “main” entity. For example, a block reference might contain two boxes, each of type \texttt{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 \texttt{AcDbFullSubentPath} is an \texttt{AcDbSubentId} object, which has a subentity type (vertex, edge, or face) and the index of the subentity in the list. Use the \texttt{AcDbSubentId} function \texttt{type()} and \texttt{index()} to access the member data.
Using the previous example, the second edge of solid1 will have its AcDbFullSubentPath as

\{(InsertID, solid1ID)
  |kEdgeSubentType, 2\};

If you have only a solid, AcDbFullSubentPath for the first face of the solid would be

\{(solidID)
  |kFaceSubentType, 1\};

**Simple Highlighting Example**

The code example later in this section shows how to highlight a sub-entity. The basic steps are as follows:

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 ADS functions. (For more information on these functions, see the ADS Developer's Guide.)

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

int ads_ssnamex(
    struct resbuf** rbpp,
    const  ads_name ss,
    const  long     i);
```

First, use `ads_ssget()` to obtain the selection set. You'll iterate through this set to obtain the individual entities it contains. Then, use `ads_ssnamex()` to obtain a subentity GS marker for the selected entity.

**Converting GS markers to subentity paths**

Use the getSubentPathsAtGsMarker() function to obtain the subentity for the GS marker returned by `ads_ssnamex()`. The complete syntax for this function is
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 getSubentPathsAtGsMarker() 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 ads_nentsel() and ads_nentselp() 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() method and pass in the subentity path. If you call highlight() 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 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;
```

```
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()` method, however, is implemented at the `AcDbEntity` level and is not generally expected to be overridden.

```cpp
// This function calls `getObjectAndGsMarker` to get the object ID of a solid and its gsmarker. Then it 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 `ads_ssget` to let the user select a single entity. It then passes this selection set to `ads_ssnamex` to get the GsMarker. Finally, the entity name in the selection set is used to obtain the object ID of the selected entity.

```cpp
Acad::ErrorStatus
getObjectAndGsMarker(AcDbObjectId& objId, int& marker)
{
    ads_name sset;
    if (ads_ssget(NULL, NULL, NULL, NULL, sset) != RTNORM) {
        ads_printf("\nads_ssget has failed");
        return Acad::eInvalidAdsName;
    }

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

    // Walk the list to the 3rd item which is the selected entity's ename.
    //
    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);
```
Chapter 6 Entities

// Move on to the 4th list element which is the GsMarker
pTemp = pTemp->rbnext;
marker = pTemp->resval.rint;

ads_relrb(pRb);
acdbGetObjectld(objId, ename);
return Acad::eOk;

// This function accepts an ObjectId 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)
{
    // space for ads_getstring pauses below
    char dummy[133];
    AcDbEntity *pEnt;
    acdbOpenAcDbEntity(pEnt, objId, AcDb::kForRead);

    // Get the subentity ID for the edge picked
    AcGePoint3d pickpnt;
    AcGeMatrix3d xform;
    int numIds;
    AcDbFullSubentPath *subentIds;
    pEnt->getSubentPathsAtGsMarker(AcDb::kEdgeSubentType, marker, pickpnt, xform, numIds, subentIds);

    // At this point the subentIds 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]

    // Objects with no edges (such as a sphere) will set
    // marker to a value less than 1 indicating there is no
    // valid subentId. In this case the code to highlight
    // an edge is meaningless and must be skipped.

    if (marker > 0) {
        // Highlight the picked edge.
        //
        pEnt->highlight(subentIds[0]);
        // Pause to let user see the effect.
        //
        ads_getstring(0, "press <RETURN> to continue...", dummy);
        // Unhighlight the picked edge.
        //
        pEnt->unhighlight(subentIds[0]);
// Get a copy of the edge, add it to the database.
//
AcDbEntity *pEntCpy = pEnt->subentPtr[subentIds[0]];
AcDbObjectId objId;
addToModelSpace(objId, pEntCpy);
}
dele te [] subentIds;
}
pEnt->close();
}

// This function accepts an ObjectId 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 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.
        //
        ads_getstring(0, "\npress <RETURN> to continue...", dummy);
        pEnt->unhighlight(subentIds[i]);
    }
dele te [] subentIds;
pEnt->close();
}
// This function accepts anobjectId. The object is opened,
// and it's highlight() and unhighlight() functions are
// used with no parameters in order 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.
    //
    ads_getstring(0, "\npress <RETURN> to continue...",
                   dummy);

    pEnt->unhighlight();
    pEnt->close();
}

Acad::ErrorStatus
addToModelSpace(AcDbObjectId &objId, AcDbEntity* pEntity)
{
    AcDbBlockTable *pBlockTable;
    AcDbBlockTableRecord *pSpaceRecord;
    acdbCurDwg()->getBlockTable(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 (\textit{ins3}) is an insert of a block that contains poly3 and box3. Insert 2 (\textit{ins2}) is an insert of a block that contains poly2, box2, and \textit{ins3}. Insert 1 (\textit{ins1}) is an insert of a block that contains poly1, box1, and \textit{ins2}.
After the inserts are created, the example highlights the different components.

```cpp
void insertHighlightCommand() {
    // Create a nested insert and try highlighting it's various sub-components.
    // There are six entities in total -- three polys and three boxes (solids). Let's call them: poly1, poly2, poly3 and box1, box2, box3. We also have three inserts: ins1, ins2, ins3.
    // ins3 is insert of block containing (poly3, box3)
    // ins2 is insert of block containing (poly2, box2, ins3)
    // ins1 is insert of block containing (poly1, box1, ins2)
    // Let's create these things first.
    // Polys
    AcDbPoly *poly1, *poly2, *poly3;
    AcGeVector3d norm(0, 0, 1);
    poly1 = new AcDbPoly(AcGePoint3d(2, 8, 0), AcGePoint3d(4, 8, 0), 6, norm, "POLY1");
    poly2 = new AcDbPoly(AcGePoint3d(7, 8, 0), AcGePoint3d(9, 8, 0), 6, norm, "POLY2");
    poly3 = new AcDbPoly(AcGePoint3d(12, 8, 0), AcGePoint3d(14, 8, 0), 6, norm, "POLY3");
    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

 ads_command(RTSTR, "_globcheck", RTSHORT, 0, RTNONE);

 ads_command(RTSTR, "BLOCK", RTSTR, "blk3", RTSTR, "0,0",
 RTSTR, "14,8", RTSTR, "11,1", RTSTR, "", RTNONE);

 ads_command(RTSTR, "INSERT", RTSTR, "blk3", RTSTR, "0,0",
 RTSHORT, 1, RTSHORT, 1, RTSHORT, 0, RTNONE);

 ads_command(RTSTR, "BLOCK", RTSTR, "blk2", RTSTR, "0,0",
 RTSTR, "9,8", RTSTR, "6,1", RTSTR, "11,1", RTSTR, "", RTNONE);

 ads_command(RTSTR, "INSERT", RTSTR, "blk2", RTSTR, "0,0",
 RTSHORT, 1, RTSHORT, 1, RTSHORT, 0, RTNONE);

 ads_command(RTSTR, "BLOCK", RTSTR, "blk1", RTSTR, "0,0",
 RTSTR, "4,8", RTSTR, "1,1", RTSTR, "6,1", RTSTR, "", RTNONE);

 ads_command(RTSTR, "INSERT", RTSTR, "blk1", RTSTR, "0,0",
 RTSHORT, 1, RTSHORT, 1, RTSHORT, 0, RTNONE);
// By now, we have the nested insert we need.  
// Grab the top most insert and open it for read and  
// hold onto it.  
//
ads_name ename;  
ads_entlast(ename);  
AcDDObjectId topInsertId;  
acdbGetObjectId(topInsertId, ename);  

// We will test highlighting of all the polys (one side  
// and the  
//)
// Then, we will do the same for all the boxes and we  
// will highlight faces of the boxes as well as the  
// whole box.  
//
// Finally, we will highlight the internal inserts.  
//
// Let's just do one right now. Do an ssget and select  
// poly. Then This is utterly ridiculous. We have to  
// write an ads_nentselx() and ads_nentselpx() that  
// returns the gsMarker. ads_ssget() is supposed to  
// return the stack with "-N" mode, but I am not sure if  
// it works.

ads_name sset;  
short marker;  
switch (ads_ssget(NULL, NULL, NULL, NULL, sset)) {
  case RTNORM:  
    // This code assumes that the user selected only one  
    // item and that's an insert. This is just some test  
    // code, so minimal (zero) error checking is being  
    // done.
    struct resbuf* rb;  
    if (ads_ssnamex(rb, sset, 0) != RTNORM) {  
      ads_ssfree(sset);  
      return;  
    }  
    int sel_method;  
    ads_name subname;  
    AcGePoint3d pickpnt;  
    AcGeVector3d pickvec;  
    AcGeMatrix3d xform;  
    struct resbuf* containers;
if (!extractEntityInfo(rb, sel_method, ename, subname, marker, pickpnt, pickvec, containers)) {
    ads_ssfree(sset);
    return;
}
ads_ssfree(sset);
break;
}
default:
    assert(Adesk::kFalse);
    break;
}
assert(marker != 0);
if (marker == 0)
    return;
pickpt[0] = 14; pickpt[1] = 8; pickpt[2] = 0;
if (ads_nentselp(NULL, ename, pickpt, TRUE, adsmat, &insStack) != RTNORM) {
    return;
}
if (insStack == NULL)
    return;
struct resbuf* rb = insStack;

// Just allocate space for 100 ids.
AcDbObjectId* idArray = new AcDbObjectId[100];
int count = 0;
AcDbObjectId objId;
acdbGetObjectId(objId, ename);
idArray[count++] = objId;
while (rb != NULL) {
    ename[0] = rb->resval.rlname[0];
    ename[1] = rb->resval.rlname[1];
    acdbGetObjectId(objId, ename);
idArray[count++] = objId;
    rb = rb->rbnext;
}
count...;
assert(count <= 3);
AcGeMatrix3d xform;
int numPaths;
AcDbFullSubentPath* subentPaths;
AcDbBlockReference* pInsert;
AcGePoint3d pickpnt;
AOK(acdbOpenObject(pInsert, topInsertId, AcDb::kForRead));

pInsert->getSubentPathsAtGsMarker(AcDb::kEdgeSubentType, marker, pickpnt, xform, numPaths, subentPaths, count, idArray);
assert(numPaths == 1);
assert(subentPaths != NULL);

AcDbFullSubentPath subPath;
AOK(pInsert->highlight(subentPaths[0]));

Adesk::Boolean yes = Adesk::kTrue, interrupted = Adesk::kFalse;
if (getYOrN("Unhighlight? [Y]", Adesk::kTrue, yes) == Acad::eOk && !interrupted && yes == Adesk::kTrue)
{
    AOK(pInsert->unhighlight(subentPaths[0]));
} else if (interrupted)
    return;

delete[] subentPaths;

// Highlight the poly.
//
subPath.objectIds().append(idArray[3]);
subPath.objectIds().append(idArray[2]);
subPath.objectIds().append(idArray[1]);
subPath.objectIds().append(idArray[0]);
pInsert->highlight(subPath);
if (getYOrN("Unhighlight? [Y]", Adesk::kTrue, yes) == Acad::eOk && !interrupted && yes == Adesk::kTrue)
    AOK(pInsert->unhighlight(subPath));
else if (interrupted)
    return;

// Highlight the insert above poly (3rd in the list).
//
subPath.objectIds().removeAt(subPath.objectIds().length() - 1);
pInsert->highlight(subPath);
if (getYOrN("Unhighlight? [Y]", Adesk::kTrue, yes) == Acad::eOk && !interrupted && yes == Adesk::kTrue)
    AOK(pInsert->unhighlight(subPath));
else if (interrupted)
    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 mtext 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 up 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.

<table>
<thead>
<tr>
<th>Entity</th>
<th>By itself</th>
<th>Nonuniform scaling (when in a block)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcDb3dSolid</td>
<td>regions, bodies</td>
<td>NA; can't be exploded</td>
</tr>
<tr>
<td>AcDbBody</td>
<td>regions, bodies</td>
<td>NA</td>
</tr>
<tr>
<td>Ac2dDbPolyline</td>
<td>lines, arcs</td>
<td>self/NA</td>
</tr>
</tbody>
</table>
### Exploding Entities (continued)

<table>
<thead>
<tr>
<th>Entity</th>
<th>By itself</th>
<th>Nonuniform scaling (when in a block)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ac3dPolyline</td>
<td>lines</td>
<td>self</td>
</tr>
<tr>
<td>AcDbArc</td>
<td>self</td>
<td>ellipse</td>
</tr>
<tr>
<td>AcDbCircle</td>
<td>self</td>
<td>ellipse</td>
</tr>
<tr>
<td>AcDbDimension</td>
<td>solids, lines, text strings, points</td>
<td>NA</td>
</tr>
<tr>
<td>AcDbEllipse</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbLeader</td>
<td>self</td>
<td></td>
</tr>
<tr>
<td>AcDbLine</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbRay</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbSpline</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbXline</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbFace</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbMLine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AcDbMText</td>
<td>one mtext for each line</td>
<td>self</td>
</tr>
<tr>
<td>AcDbPoint</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbPolyFaceMesh</td>
<td>AcDbFace</td>
<td>self</td>
</tr>
<tr>
<td>AcDbPolygonMesh</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbRegion</td>
<td>curves (splines, lines, arcs, circles)</td>
<td>NA</td>
</tr>
<tr>
<td>AcDbShape</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbSolid</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbText</td>
<td>self</td>
<td>self</td>
</tr>
<tr>
<td>AcDbTrace</td>
<td>self</td>
<td>self</td>
</tr>
</tbody>
</table>
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()` method):

- 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.

```cpp
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;
    acdbCurDwg()->getBlockTable(pBlockTable,
        AcDb::kForRead);
    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
        AcDb::kForWrite);
    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.

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

    // Get block table.
    //
    AcDbBlockTable *pBlockTable = NULL;
    acdbCurDwg()->getBlockTable(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 “AcDbEntity ownership hierarchy” on page 78).
A block can contain attribute definitions, which are templates for creating attributes. An attribute is informational text associated with a block. Depending 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 run time.

**To create a block table record, follow these steps:**

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.

```c++
void defineBlockWithAttributes(
    AcDbObjectId& blockId, // Returned value.
    const AcGePoint3d& basePoint,
    double textHeight,
    double textAngle)
{
    int retCode = 0;
    AcDbBlockTable *pBlockTable = NULL;
    AcDbBlockTableRecord* pBlockRecord = new AcDbBlockTableRecord;
    AcDbObjectId entityId;

    // Set block name and base point of the block definition
    //
    pBlockRecord->setName("ASDK-BLOCK-WITH-ATTR");
    pBlockRecord->setOrigin(basePoint);

    // Open block table for write
    //
    acdbCurDwg()->getBlockTable(pBlockTable, AcDb::kForWrite);

    // Step 2: Add 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 which 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);
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 creates a block reference rather than copying the block itself into the drawing and thus conserves memory.

If you insert a block with attribute definitions, the attribute values can be filled in by the user at run time or by the application when the block is inserted, as shown in the following example.

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 to step through the attribute definitions and fill in the values.

The following example creates a block reference, fills in the attributes, and appends the reference to the database. It uses ADS 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 attdef
    // AcGePoint3d basePoint;
    ads_getpoint(NULL, "Enter insertion point: ",
                  asDblArray(basePoint));

    // Get the rotation angle for the attribute definition
    double textAngle;
    ads_getangle(asDblArray(basePoint),
                  "Enter rotation angle: ", &textAngle);

    // Height used for the attribute definition text
    double textHeight;
    ads_getdist(asDblArray(basePoint),
                "Enter text height: ", &textHeight);

    // Build 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);
    ads_trans(&normal.x, &from, &to, Adesk::kTrue, &normal.x);

    // Set the insertion point for the block reference
    // pBlkRef->setPosition(basePoint);

    // The LCS 0.0 angle, not necessarily the UCS 0.0 angle.
    // pBlkRef->setRotation(0.0);
    pBlkRef->setNormal(normal);
// Step 3: Open current database's Model Space
// blockTableRecord

AcDbBlockTable *pBlockTable;
acdbCurDwg()->getBlockTable(pBlockTable,
   AcDb::kForRead);

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

pBlockTableRecord->close();

// Append the block reference to the model space
// block table record

AcDb ObjectId 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());

      // pBlkRef is the block reference.
      // pAttdef is the attribute definition.
      // pEnt is the entity.
      //
// Translate attribute by block reference.
// To be really correct, 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());

// Database Column value should be displayed
// INSERT would prompt for this...

pAtt->setTextString("Assigned Attribute Value");

AcDbObjectId attId;
pBlkRef->appendAttribute(attId, pAtt);
pAtt->close();
pEnt->close();
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.

```c
void printAll()
{
    char blkName[50];
    ads_getstring(Adesk::kTrue,
                  "Enter Block Name <CR for current space>: ",
                  blkName);

    if (blkName[0] == '\0') {
        if (acdbCurDwg()->tilemode() == Adesk::kFalse) {
            struct resbuf rb;
            ads_getvar("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;
    acdbCurDwg()->getBlockTable(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();
        ads_printf("Object Id %lx, handle %s, class %s.
                      Object Id\n",
                   pEntity->objectId(), handleStr, pCname);
    }
}
```
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.

```cpp
void createPolyline()
{
    // Set 4 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 4 vertex elements whose locations are supplied
    // in ptArr. The polyline has no elevation, and is
    // explicitly set as closed. The polyline is simple,
    // i.e. 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 BlockTable Object
    // AcDbBlockTable *pBlockTable;
    acdbCurDwg()->getBlockTable(pBlockTable,
        AcDb::kForRead);
}```
// Get a pointer to the MODEL_SPACE BLOCKTABLERecord
//
AcDbBlockTableRecord* pBlockTableRecord;
pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
    AcDb::kForWrite);

// Append the PLine object to the database and
// obtain its ObjId
//
AcDbObjectId plineObjId;
pBlockTableRecord->appendAcDbEntity(plineObjId, pNewPline);

// Make the PLine object reside on layer "0"
//
pNewPline->setLayer("0");

// Close them all up
//
pNewPline->close();
pBlockTableRecord->close();
pBlockTable->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 objectId of an AcDb2dPolyline, opens it, gets
// a vertex iterator, and 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();
    }
Coordinate System Access

Entity functions retrieve and set coordinate values using World Coordinate System values. The only exception to this rule is the AcDb3dPolylineVertex class, described later in this section, which uses Entity Coordinate System values. For example, if you call getCenter() 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 Entity Coordinate System (ECS). In such cases, implement the ecs() 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 ecs() function returns the identity matrix. (In other words, any time an entity’s ecs() function returns the identity matrix, you can assume the entity is defined in terms of world coordinates.)

In AutoCAD, planar entities have an Entity Coordinate System; 3D entities do not. AutoCAD entities that can return a non-identity matrix for their ecs() function are

- Dimensions
- Text
- Circles
- Arcs
- 2D polylines
- Block inserts
Points
Traces
Solids
Shapes
Attribute definitions
Attributes

**AcDb2dPolylineVertex**

An `AcDb2dPolyline` has an elevation and a series of X,Y points of class `AcDb2dPolylineVertex`. The `position()` and `setPosition()` functions of `AcDb2dPolylineVertex` specify 3D locations in the Entity Coordinate System. The Z coordinate passed in to `setPosition()` is stored in the entity and is returned by `position()` 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 methods 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.

```cpp
// Accepts an ellipse objectId, opens the ellipse and uses its getOrthoProjectedCurve member function to create a new ellipse that's 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);
}
```

```cpp
// Accepts an ellipse objectId, 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]);
}
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. It also shows 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`, 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 R12 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 non-entity object structures in a drawing. This dictionary, by default, contains two dictionaries: the group dictionary (ACAD_GROUP) and the mline style dictionary (ACAD_MLINESTYLE). 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 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 (abstract classes are shown in boldface).
An important difference between symbol tables and dictionaries is that symbol table records cannot be erased directly by an ARX 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 and/or dictionary entries (which are described in chapter 2 “Essential Database Objects”) probably will cause AutoCAD or other applications to fail.
Another important difference is that symbol table records store their associated lookup 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 cannot exceed 31 characters.
- Case is not significant.
- The character set used in names consists of 7-bit letters, digits, and the special characters dollar ($), hyphen (-), and underscore (_).
Symbol Tables

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` (For a list of the symbol table record classes, see the class hierarchy on page 131).

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).

```cpp
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

```cpp
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.
Chapter 7 Container Objects

The second version of getAt() 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 AutoCAD ARX Reference Manual.

Other important functions provided by all symbol table classes are has() and add(). See the example in “Creating and Modifying a Layer Table Record” on page 124. The signature for has() is

```cpp
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 name.

The add() function has the following signatures:

```cpp
Acad::ErrorStatus
AcDb##BASE_NAME##Table::add(AcDb##BASE_NAME##TableRecord* pRecord);
```

```cpp
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 two records by default, *MODEL_SPACE and *PAPER_SPACE, which correspond to the two spaces that are directly editable by AutoCAD users. Chapter 2, “Database Primer” and Chapter 6, “Entities” provide examples of adding entities to the model space block table record. 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 in AutoLISP and ADS with the ads_entmake() function. The BLOCK? command lists the contents of the block table, with the exception of the *MODEL_SPACE and *PAPER_SPACE records. Chapter 6, “Entities,” provides examples of creating block table records and block references. (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 write() within a program.
void AcDbLayerTableRecord::setIsLocked(Adesk::Boolean);
Adesk::Boolean AcDbLayerTableRecord::isLocked() const;
124  Chapter 7  Container Objects

**Color**

The color set by `setColor()` is used when an entity's color is BYLAYER.

```cpp
void AcDbLayerTableRecord::setColor(const AcCmColor &color);

AcCmColor AcDbLayerTableRecord::color() const;
```

**Linetype**

The linetype set by `setLinetypeObjectId()` is used when an entity's linetype is BYLAYER.

```cpp
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 linetype (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.

```cpp
void addLayer()
{
    AcDbLayerTable *pLayerTbl;
    acdbCurDwg()->getLayerTable(pLayerTbl, AcDb::kForWrite);

    if (!pLayerTbl->has("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. So, first we need to get the object id.
AcDbLinetypeTable *pLinetypeTbl;
AcDbObjectId ltId;
acdbCurDwg()->getLinetypeTable(pLinetypeTbl,
    AcDb::kForRead);
if ((pLinetypeTbl->getAt("DASHED", ltId))
    != Acad::eOk)
{
    ads_printf("Unable to find DASHED" 
               " linetype. Using CONTINUOUS");
    // CONTINUOUS is in every drawing so use it
    //
    pLinetypeTbl->getAt("CONTINUOUS", ltId);
}
else {
    pLinetypeTbl->close();
}
pLayerTblRcd->setLinetypeObjectId(ltId);
pLayerTbl->add(pLayerTblRcd);
pLayerTblRcd->close();
pLayerTbl->close();
}
else {
    ads_printf("\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 posi-
tioned 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 iter-
ator class, see the AutoCAD ARX 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 `AcDbTableRecord` class returns an object of class `AcDbTableRecordIterator` 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.

```c
void iterateLinetypes()
{
    AcDbLinetypeTable *pLinetypeTbl;
    acdbCurDwg()->getLinetypeTable(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;
    for (; !pLtIterator->done(); pLtIterator->step()) {
        pLtIterator->getRecord(pLtTableRcd, AcDb::kForRead);
        pLtTableRcd->getName(pLtName);
        pLtTableRcd->close();
        ads_printf("Linetype 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

```cpp
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() method 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 AutoCAD ARX Reference Manual.

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 = acdbCurDwg()->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 entities that are not lines. Finally, it changes the remaining entities in the group to red.

```cpp
void groups()
{
    AcDbGroup *pGroup = new AcDbGroup("grouptest");

    AcDbDictionary *pGroupDict;
    acdbCurDwg()->getGroupDictionary(pGroupDict, AcDb::kForWrite);

    AcDbObjectId groupId;
    pGroupDict->setAt("ASDK_GROUPTEST", pGroup, groupId);
    pGroupDict->close();
    pGroup->close();
    makeGroup(groupId);
    removeAllButLines(groupId);
}
```

```cpp
// Prompts the user to select objects to add to the group, then 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 = ads_ssget(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;
ads_sslength(sset, &length);
for (i = 0; i < length; i++) {
  ads_ssname(sset, i, ename);
  acdbGetObjectId(entId, ename);
  pGroup->append(entId);
}
pGroup->close();
ads_ssfree(sset);

// Accepts an object id of an AcDbGroup object, opens it,
// then iterates over the group removing all entities that
// are not AcDbLines and change all remaining entities in
// the group to color red.
//
void removeAllButLines(AcDbObjectId groupId)
{
  AcDbGroup *pGroup;
  acdbOpenObject(pGroup, groupId, AcDb::kForWrite);

  AcDbGroupIdIterator *pIter = pGroup->newIterator();
  AcDbObject *pObj;
  for (; !pIter->done(); pIter->next()) {
    pObj = pIter->getObject(AcDb::kForRead);

    // If it’s 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 mline, such as offset, color, and linetype.

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 `setAt()`.

Note: You need to close the objects after adding them with `setAt()`.

```cpp
// Creates two objects of class "AsdkMyClass" fills them in
// with the integers 1 & 2 and then adds them to the
// dictionary associated with the key "ASDK_DICT". If this
// dictionary doesn't exist, then it's created and added
// to the named objects dictionary.

void createDictionary()
{
    AcDbDictionary *pNamedobj;
    acdbCurDwg()->getNamedObjectsDictionary(pNamedobj,
        AcDb::kForWrite);
```
Dictionaries

// Check to see if the dictionary we want to create is
// already present if it's not, then 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;
    acdbCurDwg()->getNamedObjectsDictionary(pNamedobj,
        AcDb::kForRead);

    // Get a pointer to the ASDK_DICT dictionary
    //
    AcDbDictionary *pDict;
pNamedobj->getAt("ASDK_DICT", (AcDbObject*) pDict,
        AcDb::kForRead);

    pNamedobj->close();
}
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 ADS and 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:

```cpp
Acad::ErrorStatus AcDbXrecord::setFromRbChain(resbuf* pRb, AcDbDatabase* auxDb=NULL);

Acad::ErrorStatus AcDbXrecord::rbChain(resbuf** ppRb, AcDbDatabase* auxDb=NULL) const;
```

```cpp
// Get an iterator for the ASDK_DICT dictionary

AsdkMyClass *pMyCl;
Adesk::Int16 val;
for (; !pDictIter->done(); pDictIter->next()) {
    // Get the current record, open it for read and print it's data.
    //
    pDictIter->getObject((AcDbObject*&)pMyCl, AcDb::kForRead);
    pMyCl->getData(val);
    pMyCl->close();
    ads_printf("intval is:  %d", val);
}
```
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.

<table>
<thead>
<tr>
<th>DXF group code ranges for xrecords</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>140</td>
</tr>
<tr>
<td>170</td>
</tr>
<tr>
<td>210</td>
</tr>
<tr>
<td>270</td>
</tr>
<tr>
<td>280</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>310</td>
</tr>
<tr>
<td>320</td>
</tr>
<tr>
<td>330</td>
</tr>
<tr>
<td>340</td>
</tr>
<tr>
<td>350</td>
</tr>
<tr>
<td>360</td>
</tr>
</tbody>
</table>

For a description of hard and soft owners and pointers, see chapter 9, “Deriving from AcDbObject.”
Examples
The following ARX and ADS 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.

```cpp
void createXrecord()
{
    AcDbDictionary *pNamedobj, *pDict;
    acdbCurDwg()->getNamedObjectsDictionary(pNamedobj,
        AcDb::kForWrite);

    // Check to see if the dictionary we want to create is
    // already present if it's 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 = ads_buildlist(AcDb::kDxfText,
        "This is a test Xrecord list",
        AcDb::kDxfCoord, testpt,
        AcDb::kDxfReal, 3.14159,
        AcDb::kDxfAngle, 3.14159,
        AcDb::kDxfColor, 1,
        AcDb::kDxfInt16, 180,
        0);
```
// Add the data list to the Xrec. Notice that this
// member function takes a reference to resbuf *NOT* a
// pointer to resbuf, so you must dereference the the
// pointer before sending it.
pxrec->setFromRbChain(*pHead);
ads_relrb(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;
// acdbCurDwg()->getNamedObectsDictionaries(pNamedobj,
// AcDb::kForRead);
// // Get dictionary object associated with key "ASDK_DICT"
// // AcDbDictionary *pDict;
// pNamedobj->getAt("ASDK_DICT", (AcDbObject**)&pDict,
// AcDb::kForRead);
// pNamedobj->close();
// // Get the Xrec record 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);
// ads_relrb(pRbList);
// }

The following example is the ADS equivalent of the previous ARX
example.

int
createXrecord()
{
 struct resbuf *pxrec, *pDict;
 ads_point testpt = {1.0, 2.0, 0.0};
 ads_name xrecname, namedObj, dictName;
Within the named object dictionary, attempt to get the dictionary associated with the key "ASDK_DICT". If it's not present, create it.

```c
ads_namedobject(namedObj);
pDict = ads_dictsearch(namedObj, "ASDK_DICT", 0);
if (pDict == NULL) {
pDict = ads_buildlist(RTDXF0, "DICTIONARY", 100, "AcDbDictionary", 0);
    ads_entmakex(pDict, dictName);
    ads_relrb(pDict);
    ads_dictadd(namedObj, "ASDK_DICT", dictName);
}
```

Create a resbuf list of the xrecord's entity data

```c
pXrec = ads_buildlist(RTDXF0, "XRECORD", 100, "AcDbXrecord", 1, "This is a test Xrecord list", //AcDb::kDxfText 10, testpt, //AcDb::kDxfCoord 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. Place the Xrecord's new entity name into the xrecname argument.

```c
ads_entmakex(pXrec, xrecname);
ads_relrb(pXrec);
```

Set Xrecord's owner to be the ASDK_DICT dictionary.

```c
ads_dictadd(dictName, "XRECADS", xrecname);
ads_retvoid();
return RTNORM;
```

Accesses the xrecord associated with the key "XRECADS" in the dictionary associated the key "TEST_DICT". Then lists out the contents of this xrecord using the printList function.

```c
int listXrecord()
{
    struct resbuf *pObj;
    ads_name dictName;
```
// Get the entity name of the ASDK_DICT dictionary in
// the named object dictionary and place it in dictName.
//
ads_namedobjdict(dictName);
pObj = ads_dictsearch(dictName, "ASDK_DICT", 0);
if (pObj) {
    // Entity name is always returned in first resbuf.
    //
    ads_name_set(pObj->resval.rlname, dictName);
}

// Get the Xrecord associated with the key "XRECADS".
//
pObj = ads_dictsearch(dictName, "XRECADS", 0);
if (pObj) {
    printList(pObj);
    ads_relrb(pObj);
}
ads_retvoid();
return RTNORM;
DEFINING NEW CLASSES

CHAPTER 8 ■ DEFINING A CUSTOM ARX CLASS

CHAPTER 9 ■ DERIVING FROM ACDBOBJECT

CHAPTER 10 ■ DERIVING FROM ACDBENTITY
This chapter describes how to use the ARX macros to simplify the task of deriving a custom ARX class. These macros allow a custom class to participate in the AcRxObject run-time type identification mechanism. If you do not need to distinguish your custom class at run time, you can use standard C++ derivation style to create the new class.
ARX provides a set of macros, declared in the rxboiler.h file, that help you create new classes derived from AcRxObject. You can derive new classes from most of the classes in the ARX hierarchy except the AutoCAD Release 12 entity set (listed in chapter 6, “Entities”) and the symbol table classes. If you do not use the ARX macros to define your new class, the class will inherit the run-time identity of its most immediate ARX-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

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
Run-Time Class Identification

Every class in the ARX hierarchy that is derived from \texttt{AcRxObject} has a corresponding class descriptor object, which is an instance of \texttt{AcRxClass} that holds information for run-time type identification. The class descriptor object, \texttt{gpDesc}, is a static data member of the class—for example, \texttt{AcDbEllipse::gpDesc}. Class descriptor objects are created at initialization, when classes are registered with ARX and are added to a system-level dictionary, \texttt{acrxClassDictionary}. The macros described here facilitate the declaration and implementation of certain functions related to run-time identification and initialization functions, including the class initialization routine as well as the \texttt{desc()}, \texttt{cast()}, and \texttt{isA()} functions for the custom class. (See chapter 1, “Overview,” for a brief description of the \texttt{AcRxObject} functions related to run-time type identification.)

Class Declaration Macro

The header file for a custom class can use the \texttt{ACRX_DECLARE_MEMBERS(CLASS_NAME)} ARX macro to declare the \texttt{desc()}, \texttt{cast()}, and \texttt{isA()} functions.

This macro is used in the public section of the class declaration, as follows:

```cpp
class myClass : public AcRxObject
{
public:
    ACRX_DECLARE_MEMBERS(myClass);
};
```

For \texttt{AsdkPoly}, the following line expands to a single long line of code.

```cpp
ACRX_DECLARE_MEMBERS(AsdkPoly);
```
When reformatted to multiple lines for clarity, the line looks like this:

```cpp
virtual AcRxClass* isA() const;
static AcRxClass* gpDesc;
static AcRxClass* desc();
static AsdkPoly* cast(const AcObject* 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, VERN0)**
  
  Use for transient classes that can be instantiated but are not written to file.

- **ACRX_DXF_DEFINE_MEMBERS(CLASS_NAME, PARENT_CLASS, VERN0, 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 146)
- **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:

```cpp
ACRX_DXF_DEFINE_MEMBERS(AsdkPoly, AcDbCurve, 0, POLYGON, /*MSG0*/"AutoCAD");
```

When reformatted to multiple lines for clarity, the line looks like this:

```cpp
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", "AcDbCurve", 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. So, this semicolon is not required for this macro call line.

If you want to write your own \texttt{rxInit()} function, use the \texttt{ACRX\_DEFINE\_MEMBERS()} macro by itself, which defines \texttt{desc()}, \texttt{cast()}, and \texttt{isA()} for your class but does not define the \texttt{rxInit()} function. This macro also does not create the associated \texttt{AcRxClass} object, which is the responsibility of the \texttt{rxInit()} function.

### Class Initialization Function

The class initialization function for each class is \texttt{rxInit()}. An application that defines a custom class must invoke this function during run-time initialization.

This function is defined automatically by each of the three \texttt{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 \texttt{rxInit()} function, use the \texttt{ACRX\_DEFINE\_MEMBERS()} macro.
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; hard and soft pointers), and the undo and redo operations. This chapter also describes zombies—objects that are created when a file containing application-defined entities is read in but the application itself is not loaded.

The descriptions in this chapter assume you are familiar with the material presented in chapter 5, “Database Objects” and chapter 8, “Deriving a custom ARX Class”
Overview

If you’re subclassing from AcDbObject, there are a number of virtual functions that 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**

```cpp
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;
```

**AcDbObject: Functions Often Overridden**

```cpp
virtual Acad::ErrorStatus audit(AcDbAuditInfo*);
// Very Commonly Useful, as it happens at a point where a new // object state is being committed.
//
virtual Acad::ErrorStatus subClose();
// Next two really apply to container objects. Simple entity // is easier.
//
```

```cpp
virtual Acad::ErrorStatus deepClone(AcDbObject* pOwnerObject, 
    AcDbObject*& pClonedObject, 
    AcDbIdMapping& idMap, 
    Adesk::Boolean isPrimary = Adesk::kTrue) const;
```

```cpp
virtual Acad::ErrorStatus wblockClone(AcRxObject* pOwnerObject, 
    AcDbObject*& pClonedObject, 
    AcDbIdMapping& idMap, 
    Adesk::Boolean isPrimary = Adesk::kTrue) const;
```
Overview 149

AcDbObject: Functions Sometimes Overridden

```cpp
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

```cpp
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 void 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**

// Dangerous, and Rarely Useful to override;
// an AcRxObject members

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**

// NOTE: ALL AcDbObject virtual members can apply here as well!

virtual Acad::Boolean worldDraw(AcGiWorldDraw* mode);

virtual void viewportDraw(AcGiViewportDraw* mode);

virtual Acad::ErrorStatus getGeomExtents(AcDbExtents& extents) const;

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

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**

// Typically done, although not essential
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);

// Common (i.e. not for every entity type)
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, AcDbObject* & entAndInsertStack = NULL) const;

virtual Acad::ErrorStatus applyPartialUndo(AcDbDwgFiler* undoFiler, AcRxClass* classObj);

virtual void subSetDatabaseDefaults(AcDbDatabase* pDb);

// Uncommon
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 void saveAs(AcGiWorldDraw* mode, AcDb::SaveType st);

virtual Adesk::Boolean saveImagesByDefault() const;

// Rarely (if ever) useful.
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);

AcDbEntity: Functions Rarely Overridden

// Rarely (if ever) useful.
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);
**AcDbCurve: Functions to Override**

```cpp
virtual Adesk::Boolean
isClosed () const;

virtual Adesk::Boolean
isPeriodic () const;

virtual Adesk::Boolean
isPlanar () const;

virtual Acad::ErrorStatus
getPlane (AcGePlane&, Acad::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;
```
Chapter 9 Deriving from Acad\n
```cpp
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& area) 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. (For more information, see “Undo and Redo” on page 178.) Even if you don’t desire undo recording, it is essential to call

```c
assertWriteEnabled(kFalse, kFalse);
```

This call marks the object for incremental save. Failure to follow this instruction can result in corrupt 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.

<table>
<thead>
<tr>
<th>Object open for</th>
<th>Read</th>
<th>Write</th>
<th>Notify</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertReadEnabled()</code></td>
<td>returns</td>
<td>returns</td>
<td>returns</td>
</tr>
<tr>
<td><code>assertWriteEnabled()</code></td>
<td>aborts</td>
<td>returns</td>
<td>aborts</td>
</tr>
<tr>
<td><code>assertNotifyEnabled()</code></td>
<td>returns</td>
<td>returns</td>
<td>returns</td>
</tr>
</tbody>
</table>
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.

```cpp
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 `ads_entmake()`, `ads_entmod()`, and `ads_entget()`)
- `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 `kPurgeFiler`)
- Any time an object is paged out (uses `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 `kFileFiler`)
- **UNDO** (uses `kUndoFiler`)
- **INSERT, COPY, XREF** (use `kDeepCloneFiler` and a `kIdXlateFiler`)
- **WBLOCK** (uses `kWblockCloneFiler` and `kIdXlateFiler`)
- Any time an object is paged in (uses `kPageFiler`)

### dwfOut() Function

The `dwfOut()` function, which calls `dxfOutFields()`, is invoked by the following commands and functions:

- **DXFOUT**
- `ads_entget()`
Chapter 9 Deriving from AcDbObject

**dxfIn() Function**

The `dxfIn()` function, which calls `dxfInFields()`, is invoked by the following commands and functions:

- DXFIN
- `ads_entmod()` or `ads_entmake()`

**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:

```cpp
AcDbDerivedClass::dwgOutFields(...);
{
    assertReadEnabled();
    myParent::dwgOutFields();
    // Class-specific operations after supermessaging
}
```

If you forget to call the corresponding message of the parent class, you’ll receive a run-time 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()`:

```cpp
Acad::ErrorStatus AsdkPoly::dwgOutFields(AcDbDwgFiler* filer) const
{
    assertReadEnabled();
    Acad::ErrorStatus es;
    if ((es = AcDbCurve::dwgOutFields(filer)) != Acad::eOk)
    {
        return es;
    }

    filer->writeItem(mCenter);
    filer->writeItem(mStartPoint);
    filer->writeInt32(mNumSides);
    filer->writeItem(mPlaneNormal);
    filer->writeItem(mpName);
    // mTextStyle is a hard pointer id, so filing it out to
    // the purge filer (kPurgeFiler) prevents purging of
    // this object.
    //
    // filer->writeItem(mTextStyle);
    return es;
}
```

Sample Code for dwgInFields()

The following is sample code for `AsdkPoly::dwgInFields()`:

```cpp
Acad::ErrorStatus AsdkPoly::dwgInFields(AcDbDwgFiler* filer)
{
    assertWriteEnabled();
    Acad::ErrorStatus es;
```
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()`. Then, it must super-message (that is, call the same function on the parent class).

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.

<table>
<thead>
<tr>
<th>DXF group code ranges for object representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>140</td>
</tr>
<tr>
<td>170</td>
</tr>
</tbody>
</table>
An object ID translates to a 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():

```cpp
Acad::ErrorStatus AsdkPoly::dxfOutFields(AcDbDxfFiler* filer) const
{
    assertReadEnabled();
    Acad::ErrorStatus es;
    // Code here...
}
```

DXF group code ranges for object representation (continued)

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>210</td>
<td>219</td>
<td>3 reals</td>
</tr>
<tr>
<td>270</td>
<td>279</td>
<td>16-bit integer</td>
</tr>
<tr>
<td>280</td>
<td>289</td>
<td>8-bit integer</td>
</tr>
<tr>
<td>300</td>
<td>309</td>
<td>text</td>
</tr>
<tr>
<td>310</td>
<td>319</td>
<td>binary chunk</td>
</tr>
<tr>
<td>320</td>
<td>329</td>
<td>handle</td>
</tr>
<tr>
<td>330</td>
<td>339</td>
<td>soft pointer ID</td>
</tr>
<tr>
<td>340</td>
<td>349</td>
<td>hard pointer ID</td>
</tr>
<tr>
<td>350</td>
<td>359</td>
<td>soft owner ID</td>
</tr>
<tr>
<td>360</td>
<td>369</td>
<td>hard owner ID</td>
</tr>
</tbody>
</table>
if ((es = AcDbCurve::dxflnFields(filer)) != Acad::eOk)
{
    return es;
}
filer->writeItem(AcDb::kDxfSubclass, "AsdkPoly");
filer->writeItem(AcDb::kDxfXCoord, mCenter);
filer->writeItem(AcDb::kDxfXCoord + 1, mStartPoint);
filer->writeInt32(AcDb::kDxfInt32, mNumSides);
filer->writeItem(AcDb::kDxfNormalX, mPlaneNormal);
filer->writeItem(AcDb::kDxfText, mpName);
filer->writeItem(AcDb::kDxfHardPointerId, mTextStyle);
return es;

Sample Code for dxflnFields() with Order Independence

The following is sample code for AsdkPoly::dxflnFields():

Acad::ErrorStatus
AsdkPoly::dxflnFields(AcDbDxfFiler* filer)
{
    assertWriteEnabled();
    Acad::ErrorStatus es = Acad::eOk;
    resbuf rb;

    if ((AcDbCurve::dxflnFields(filer) != Acad::eOk)
        || !filer->atSubclassData("AsdkPoly"))
    {
        return filer->filerStatus();
    }

    while ((es == Acad::eOk)
        && (es = filer->readResBuf(&rb)) == Acad::eOk)
    {
        switch (rb.restype) {
            case kDxfXCoord:
                mCenter = asPnt3d(rb.resval.rpoint);
                break;
            case kDxfXCoord + 1:
                mStartPoint = asPnt3d(rb.resval.rpoint);
                break;
            case kDxfInt32:
                mNumSides = rb.resval.rlong;
                break;
            case kDxfNormalX:
                mPlaneNormal = asVec3d(rb.resval.rpoint);
                break;
        }
    }
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.

```cpp
Acad::ErrorStatus AsdkPoly::dxfInFields(AcDbDxfFiler* filer)
{
    assertWriteEnabled();
    if ((AcDbCurve::dxfInFields(filer) != Acad::eOk) ||
        !filer->atSubclassData("AsdkPoly"))
    {
        return filer->filerStatus();
    }

    if (es != Acad::eEndOfFile)
    return Acad::eInvalidResBuf;

    return es;
}
```
struct resbuf rb;
filer->readItem(&rb);
if (rb.restype == AcDb::kDxfXCoord) {
mCenter.set(rb.resval.rpoint[X],
           rb.resval.rpoint[Y], rb.resval.rpoint[Z]);
} else {  
    filer->pushBackItem();
    filer->setError(Acad::eInvalidDxfCode,
                    "%Error: expected group code %d",
                    AcDb::kDxfXCoord);
    return filer->filerStatus();
}

filer->readItem(&rb);
if (rb.restype == AcDb::kDxfXCoord + 1) {  
mStartPoint.set(rb.resval.rpoint[X],
                rb.resval.rpoint[Y], rb.resval.rpoint[Z]);
} else {  
    filer->pushBackItem();
    filer->setError(Acad::eInvalidDxfCode,
                    "%Error: expected group code %d",
                    AcDb::kDxfXCoord + 1);
    return filer->filerStatus();
}

filer->readItem(&rb);
if (rb.restype == AcDb::kDxfInt32) {  
mNumSides = rb.resval.rlong;
} else {  
    filer->pushBackItem();
    filer->setError(Acad::eInvalidDxfCode,
                    "%Error: expected group code %d",
                    AcDb::kDxfInt32);
    return filer->filerStatus();
}

filer->readItem(&rb);
if (rb.restype == AcDb::kDxfNormalX) {  
mPlaneNormal.set(rb.resval.rpoint[X],
                rb.resval.rpoint[Y], rb.resval.rpoint[Z]);
} else {  
    filer->pushBackItem();
    filer->setError(Acad::eInvalidDxfCode,
                    "%Error: expected group code %d",
                    AcDb::kDxfNormalX);
    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.

The ownership connection is usually a two-way link that requires two steps:

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 as well as 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:

```c++
// Uses the OwnerDemo class defined in the next example
// (see “ARX Example,” below)
// Sets pOwner to be the owner of pOwned
void makeOwner(OwnerDemo* pOwner, AcDbObject* pOwned) {
    // First let pOwner know its the owner. This establishes ownership for filing persistence.
    pOwner->setIdData(pOwned->objectId());

    // Now set up the backpointer so that the owned object knows 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 both 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.

```c++
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 ADS or 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 13, “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 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, 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, first using ARX functions, and then using ADS and AutoLISP 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.
ARX Example

// class declarations

class AsdkOwnerDemo : public AcDbObject

// This is a custom object class to demonstrate what's 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 objectId 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, 0, 
ASDKOWNERDEMO, OWNERSHIP);
MAKE_ACDBOPENOBJECT_FUNCTION(AsdkOwnerDemo);

// Gets value of integer data member

Adesk::Int16 AsdkOwnerDemo::intData()
{
    assertReadEnabled();
    return mIntval;
}

// Sets value of integer data member

Acad::ErrorStatus AsdkOwnerDemo::setIntData(const Adesk::Int16& val)
{
    assertWriteEnabled();
    mIntval = val;
    return Acad::eOk;
}
Chapter 9 Deriving from AcDbObject

// 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 that 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);
    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);
    filer->writeItem(mIntval);
    filer->writeItem(mObjId);
    return filer->filerStatus();
}
ifdef 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();
}

ifdef 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 objectIds 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. Next 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 objects dictionary. Technically we could just add pObjA to the named objects dictionary itself, but that's bad form since it will tend to clutter up the named objects dictionary.

void createObjs()
{
    AcDbObjectId objIdA, objIdB, objIdC;
    AcDbDictionary *pNamedobj;
    AcDbDictionary *pDict = NULL;
    AcDbDatabase *pCurDwg = acdbCurDwg();

    // Create object C with dummy integer data value of 3
    AsdkOwnerDemo *pObjC = new AsdkOwnerDemo(3);
    pObjC->setIdData(3);
    pObjC->close();

    // Append object C to database without setting an owner
    pCurDwg->addAcDbObject(objIdC, pObjC);

    // Create object B with dummy integer data value of 2
    AsdkOwnerDemo *pObjB = new AsdkOwnerDemo(2);
    pObjB->setIdData(2);
    pObjB->close();

    // Append object B to 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 dummy integer data value of 1
    AsdkOwnerDemo *pObjA = new AsdkOwnerDemo(1);
    pObjA->setIdData(1);
    pObjA->close();
}
Ownership References

// Next add objA to a dictionary in the named objects
// 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();

// 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;
   acdbCurDwg()->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);
    ads_printf("Intdata: %d  ObjId: %ld  Backpointer: %ld
    (pDemo->objectId()).asOldId(),
    (pDemo->ownerId()).asOldId(),
    (pDemo->idData()).asOldId());

    // Recursive tree walk
    if ((pDemo->idData()).asOldId() != 0L) {
        printOut(pDemo->idData());
    }
    pDemo->close();
}

// Initialization function 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();
}

// Clean up function called from acrxEntryPoint during the // kUnloadAppMsg case. This function removes this apps // command set from the command stack and removes 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 database is // still active, it should cause all objects of class // AsdkOwnerDemo to be zombified. //
deleteAcRxClass(AsdkOwnerDemo::desc());

// ARX entry point //
AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt) {
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(pkt);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
    }
    return AcRx::kRetOK;
}

**ADS/AutoLISP Example**

This code example shows how to build the same ownership hierarchy as in the previous example, this time using ADS/AutoLISP function calls. Again, you need to add each object to the database to obtain an ads_name, which is the ADS equivalent of a C++ object ID. This example shows only the code for the createObjs() routine.

```lisp
int createObjs()
{
    struct resbuf *pList;
    ads_name enameA, enameB, enameC,
    namedObj, enameTestDict;

    // Create a resbuf list of objC's info and data. This
    // will include a 'dummy' ownership group of 0 since
    // this object won't own any others. The group 70 is the
    // 'dummy' integer data value.
    // pList = ads_buildlist(RTDXF0, "ASDKOWNERDEMO", 100,
    // "AsdkOwnerDemo", 70, 3, 0);

    // Create the object 'C' with no owner set. The
    // object's new entity name will be placed into the
    // objC argument
    // ads_entmakex(pList, enameC);
    ads_relr(pList);
```
// Create a resbuf list of objB's info and data. This
// will include an ownership group with the entity name
// of objC. The group 70 is the "dummy" integer data
// value.
//
// pList = ads_buildlist(RTDXF0, "ASDKOWNERDEMO",
// 100, "AsdkOwnerDemo", 70, 2, 360, enameC, 0);

// Create the object "B" with no owner set. The
// object's new entity name will be placed into the objC
// argument
//
// ads_entmakex(pList, enameB);
// ads_relrb(pList);

// Create a resbuf list of objA's info and data. This
// will include an ownership group with the entity name
// of objB. The group 70 is the "dummy" integer data
// value.
//
// pList = ads_buildlist(RTDXF0, "ASDKOWNERDEMO",
// 100, "AsdkOwnerDemo", 70, 1, 360, enameB, 0);

// Create the object "A" with no owner set. The
// object's new entity name will be placed into the objC
// argument
//
// ads_entmakex(pList, enameA);
// ads_relrb(pList);

// Add object A to a dictionary so it has an owner

// Look for the dictionary associated with the key
// "ASDK_DICT". If it's not found, create it and add
// it to the named objects dictionary
//
// ads_namedobjdict(namedObj);
if ((pList = ads_dictsearch(namedObj, "ASDK_DICT", 0))
== NULL) {
    pList = ads_buildlist(RTDXF0, "DICTIONARY",
                   100, "AcDbDictionary", 0);
    ads_entmakex(pList, enameTestDict);
    ads_dictadd(namedObj, "ASDK_DICT", enameTestDict);
} else {
    // ename is always returned in the first resbuf
    // ads_name_set(enameTestDict, pList->resval.rlname);
}

// Now add object A to the ASDK_DICT dictionary
// so it has an owner.
//
// ads_dictadd(enameTestDict, "OBJA", enameA);
// ads_relrb(pList);

ads_retvoid();
return RTNORM;
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 pointers are:

- 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 entity has a hard pointer reference to a layer
- 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 are:

- 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.
Chapter 9 Deriving from AcDbObject

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()` method of `AcDbDatabase` is invoked on the set of objects specified in the ID array:

```cpp
AcDbDatabase::purge(AcDbObjectIdArray &idArray);
```

The `purge()` method 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 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 methods 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:

```cpp
void assertWriteEnabled(
    Adesk::Boolean autoUndo = Adesk::kTrue,
    Adesk::Boolean recordModified = Adesk::kTrue);
```

When a modification method 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 methods of the class. If only a small portion of an object’s state is typically modified in a particular member function, 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 180.

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 `writeXXX()` function) to save the relevant information in the undo file.
Chapter 9 Deriving from AcDbObject

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.

```cpp
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`.

```cpp
Acad::ErrorStatus
AsdkPoly::setNumSides(int numSides)
{
    if (mNumSides == numSides)
        return Acad::eOk;

    assertWriteEnabled(Adesk::kFalse, Adesk::kTrue);

    // 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.

```cpp
assertWriteEnabled(kFalse, kFalse);
```
As an example, here is `AsdkPoly::applyPartialUndo()` function:

```cpp
Acad::ErrorStatus
AsdkPoly::applyPartialUndo(AcDbDwgFiler* filer,
                           AcRxClass*    classObj)
{
    // First thing we do is see if the class matches with
    // ours. If it doesn't, we call base class's
    // applyPartialUndo(), and, 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 method, you are responsible for recording for redo in your undo operation. This is usually accomplished by calling the appropriate `set()` methods. When your `set()` method 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:

```cpp
virtual Acad::ErrorStatus subErase(Adesk::Boolean pErasing);
```

If you override a subsidiary function, your function should follow these general steps:

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 bother passing the message up, because your bad error status will effectively kill the operation.

2. If everything is OK, then invoke `YourParent::subErase()`. 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 implementing a `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.

```cpp
class AsdkEllipse : public AcDbEllipse
{ // This class extends AcDbEllipse by adding in functionality
  // to store a dynamic array of hard pointer ObjectIds.
  // The subErase() member function has been overridden and
  // implemented such that whenever an object of this class is
  // erased, the object's pointed to by the hardpointer ids
  // stored within the object will also be erased.

public:
  ACRX_DECLARE_MEMBERS(AsdkEllipse);
  AsdkEllipse() {};
  AsdkEllipse(const AsdkEllipse&);
  AsdkEllipse(const AcDbObjectIdArray& elliapses)
    : mEllipseIds(elliapses) {};
```
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 setEllipses(  
const AcDbObjectIdArray& Ids);

Acad::ErrorStatus appendId(const AcDbObjectId& objId);

Acad::ErrorStatus appendIds(  
const AcDbObjectIdArray& objIds);

Acad::ErrorStatus removeId(  
const AcDbObjectId& objId);

// AcDbObject overrides
//
virtual Acad::ErrorStatus subErase(  
Adesk::Boolean pErasing);

virtual Acad::ErrorStatus dwgInFields(  
AcDbDwgFiler* filer);

virtual Acad::ErrorStatus dwgOutFields(  
AcDbDxfFiler* 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
//
// AcRxObject overrides
//
virtual AcRxObject* clone() const;

private:
    AcDbObjectIdArray mEllipseIds;
    static int mInFlux;  // == 1 when first object's  
    //    subErase is erasing
};

ACRX_DXF_DEFINE_MEMBERS(AsdkEllipse, AcDbEllipse, 0,  
ASDKELLIPSE, REFERENC);

MAKE_ACBOPENOBJECT_FUNCTION(AsdkEllipse);
// 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) {
    if  (which > mEllipseIds.length())
        return AcDbObjectId::kNull;
    return mEllipseIds[which];
}

Acad::ErrorStatus
AsdkEllipse::setEllipseId(const AcDbObjectId& objId,
    unsigned short which) {
    if  (which > mEllipseIds.length())
        return Acad::eInvalidIndex;
    mEllipseIds[which] = objId;
    return Acad::eOk;
}
subErase, subOpen, subClose, and subCancel

```cpp
Acad::ErrorStatus
AsdkEllipse::setEllipseIds(const AcDbObjectIdArray& objIds)
{
    if (objIds.length() == 0)
        return Acad::eNullObjectId;
    mEllipseIds = objIds;
    return Acad::eOk;
}

Acad::ErrorStatus
AsdkEllipse::appendId(const AcDbObjectId& objId)
{
    if (objId == AcDbObjectId::kNull)
        return Acad::eNullObjectId;
    mEllipseIds.append(objId);
    return Acad::eOk;
}

Acad::ErrorStatus
AsdkEllipse::appendIds(const AcDbObjectIdArray& objIds)
{
    if (objIds.length() == 0)
        return Acad::eNullObjectId;
    mEllipseIds.append(objIds);
    return Acad::eOk;
}

inline Adesk::Boolean
AsdkEllipse::removeId(const AcDbObjectId& objId)
{
    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);
  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();
subErase, subOpen, subClose, and subCancel

```
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.retype == 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.retype == AcDb::kDxfHardPointerId) {
        acdbGetObjectById(objId, inbuf.resval.rname);
        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());
```
Chapter 9  Deriving from AcDbObject

```cpp
for (int i = 0; i < mEllipseIds.length(); i++) {
    filer->writeObjectId(AcDb::kDxfHardPointerId, mEllipseIds[i]);
}
return filer->filerStatus();
}

void AsdkEllipse::list() const
{
    AcDbEllipse::list();
    ads_printf("Class: %s", isA()->name());
    for (int i = 0; i < mEllipseIds.length(); i++) {
        ads_printf("ReferencedId[%d]: %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
{
    return new AsdkEllipse(*this);
}

Acad::ErrorStatus AsdkEllipse::wblockClone(AcRxObject* pOwnerObject,
           AcDbObject*& pClonedObject,
           AcDbIdMapping& idMap,
           Adesk::Boolean isPrimary) const
{
    static AcDbObjectId btr;
    AcTransaction *pTrans = NULL;
    // Have we already done this entity?
    //
    AcDbIdPair idPair(objectId(), AcDbObjectId::kNull, 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);
actrTransactionManager->endTransaction();
ads_printf("WblockClone error status: %d", 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
    // getTransformedCopy uses this->clone() to create
    // a new object which the ent pointer is assigned
    // to point to. So, 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 in order to
    // be sure that an AsdkEllipse is created instead
    // of just an AcDbEntity.
    AsdkEllipse *pNextEllipse;
    ((AsdkEllipse*)ellipses[0])->getTransformedCopy(
      scale, (AcDbEntity*&)pNextEllipse);
ellipses.append(pNextEllipse);
}

AcDbBlockTable *pBlockTable;
AcdbCurDwg()->getBlockTable(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);
((AsdkEllipse*)ellipses[i])->close();
}
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 objectid 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();
acrBuildClassHierarchy();
}
void unloadApp()
{
    acedRegCmds->removeGroup("ASDK_ELLIPSES");

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

extern "C" AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(pkt);
        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. Because it is written to and read from file, 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
{
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.

class AsdkMyClass : public AcDbObject
//
// This is a custom object class to demonstrate what's
// necessary to make it work.
//
// 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;
};
ACRX_DXF_DEFINE_MEMBERS(AsdkMyClass, AcDbObject, 0,
ASDKMYCLASS, SAMP2);
MAKE_ACDBOPENOBJECT_FUNCTION(AsdkMyClass);

// Gets value of integer data member
//
Acad::ErrorStatus AsdkMyClass::getData (Adesk::Int16& val)
{
  // Tell AutoCAD a read operation is taking place.
  //
  assertReadEnabled();
  val = mIntval;
  return Acad::eOk;
}
Example of a Custom Object Class

```cpp
// Sets value of integer data member
//
Acad::ErrorStatus
AsdkMyClass::setData(Adesk::Int16 val)
{
  // Trigger 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);
  pFiler->readItem(&mIntVal);
  return pFiler->filerStatus();
}

// Files data out to a DWG file
//
Acad::ErrorStatus
AsdkMyClass::dwgOutFields(AcDbDwgFiler* pFiler) const
{
  assertReadEnabled();

  AcDbObject::dwgOutFields(pFiler);
  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();
}

// Creates two objects of class "AsdkMyClass" fills them in with the integers 1 & 2 and then adds them to the dictionary associated with the key "ASDK_DICT". If this dictionary doesn't exist, then it's created and added to the named objects dictionary.
//
void
createDictionary()
{
    AcDbDictionary *pNamedobj;
    acdbCurDwg()->getNamedObjectsDictionary(pNamedobj, AcDb::kForWrite);
    // Check to see if the dictionary we want to create is already present if it's not, then 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);
Example of a Custom Object Class

```cpp
AcDbObject 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 it's entries printing out the
// integer data value in each entry.
//
// void iterateDictionary()
{

AcDbDictionary *pNamedobj;
acdbCurDwg()->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 it's data.
    //
    pDictIter->getObject((AcDbObject*)pMyCl,
        AcDb::kForRead);
pMyCl->getData(val);
pMyCl->close();
    ads_printf("intval is:  %d", val);
}
delete pDictIter;
pDict->close();
}
```
Chapter 9 Deriving from AcDbObject

// Initialization function 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_DICTIONARY_COMMANDS",
                            "ASDK_CREATE", "CREATE", ACRX_CMD_MODAL,
                            createDictionary);

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

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

// Clean up function called from acrxEntryPoint during the
// kUnloadAppMsg case. This function removes this apps
// command set from the command stack and removes this apps
// 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 database is
    // still active, it should cause all objects of class
    // AsdkMyClass to be zombified.
    //
    deleteAcRxClass(AsdkMyClass::desc());
}

Zombies

A zombie is an object that is created automatically when AutoCAD
tries to instantiate a custom class from a DWG file—for example,
when a drawing is loaded—and the application defining the class is
not loaded. Zombies are created in memory when the defining applica-
tion is missing, and they are also written out to a file in one case.

Zombies are created in memory under the following circumstances:

- A drawing is loaded and the application defining the custom
g entity is not present
- The application that defines the custom entity is unloaded in
  mid-session
- A DXF file containing zombies is read in
When a DXF file containing custom entities is read in and the application is missing, zombies are not created. The custom entities are simply discarded. This situation can cause serious problems if your application depends on these custom entities.

Zombies are created in a file when a zombie exists in memory and a DXF file is written out.

You can never have an instance of a zombie in a DWG file.

If a custom object had hard ownership or pointer references to other objects before it became a zombie, the zombie honors the references and protects them from purge.

Displaying a Zombie Entity

When an entity is saved, the SAVEIMAGES system variable is used to determine whether an alternate graphical representation of the entity is saved along with the standard save data in the DWG file. If SAVEIMAGES is set to 1, the alternate representation is saved. If SAVEIMAGES is set to 0, the application decides whether its custom entities should save their alternate graphical representation. If SAVEIMAGES is set to 2, the alternate representation is never saved. The AcDbEntity::saveImagesByDefault() function returns kTrue by default, which means that the alternate graphical representation is saved. New custom entity classes can override this function so that the default is not to create the alternate graphical representation.

This internal representation, sometimes referred to as a graphics metafile, is used to display the entity when the defining application is missing and a zombie is created for the entity. When you supply a worldDraw() function for your custom entity class, the graphics metafile is created automatically if SAVEIMAGES indicates that it should be created (see chapter 10, "Deriving from AcDbEntity").

Saving Zombies to a DXF File

Zombies usually reside only in memory. The only case when a zombie is written out is when a DWG file is read in, zombies are created because the application is missing, and the file is subsequently saved to a DXF file. In this case, the DXF file contains instances of class AcDbZombieObject or AcDbZombieEntity.
Resurrecting Zombies

Zombies are resurrected—that is, turned back into “real” custom objects or entities—when the missing application is subsequently loaded.

**Note** Zombies are not resurrected when a DXF file containing zombies is read in (using the DXFIN command). Even if the defining application is already loaded, the zombies in the DXF file remain zombies until the application is reloaded (which then causes the zombies to “wake up”).

Unloading an Application

When an application is unloaded and the appropriate cleanup operations have been performed, custom objects and entities are transformed into zombies. At unload time, all custom classes must have been removed from the class dictionary with the `deleteAcRxClass()` function. For a description of the requirements for making an application “unloadable,” see chapter 3, “Building an ARX Application.”

Summary of Zombie Behavior

Following is a summary of what happens to application objects and zombies when DXF files are read in and filed out and when a drawing is opened and saved.

**DXFIN**

When a DXF file is read in, the following can occur:

- If the relevant application is loaded, application objects/entities are read in as themselves
- If the relevant application is not loaded, application objects/entities are dropped entirely
- Zombie objects/entities in DXF files are read in as zombies

**DXFOUT**

When a DXF file is filed out, the following can occur:

- Zombies in memory are written out as zombies
- Application objects/entities are written out as themselves (the application must be loaded if there are application objects and entities in memory)
OPEN
When a drawing is opened, the following can occur:

- If the application is loaded, application objects/entities are read in as themselves
- If the application is not loaded, application objects/entities are read in as zombies

Note There can never be any zombies in a DWG file (see the following section, “SAVE”).

SAVE
When a drawing is saved, the following can occur:

- Zombies are written out as the real application objects/entities they once were
- Application objects/entities are written out as themselves (the application must be loaded if there are application objects/entities in memory)

Object Version Support

ARX does not currently provide explicit support for object versioning, but if you plan to evolve your custom object classes, you can build in your own versioning support. There are several ways you might accomplish this task:

- Introduce an Adesk::UInt8 as the first data member of the class that represents the version of the class and file it out in the filer methods of the class. Because this data is persistent, it can be checked to determine the version of the object.
- Maintain the version number as extended data in xdata or in the extension dictionary.
This chapter describes how to derive a custom class from AcDbEntity. To create a custom entity class, you’ll need to be familiar with the AcGi library, which provides the graphics interface classes for displaying the entity. You will also need to override a number of entity functions, including worldDraw(). Specific examples of overriding virtual methods provided by the AcDbEntity class are included. For a complete listing of the custom entity class AsdkPoly, see the samples directory. Common entity operations such as object snap points, grip points, and stretch points are also discussed in further detail in this chapter.

The descriptions in this chapter assume you are familiar with the material presented in chapter 6, “Entities”; chapter 8, “Deriving a custom ARX Class”; and chapter 9, “Deriving from AcDbObject”
Displaying the Entity

The AcGi library is used to display custom entities. For a complete description of all AcGi classes and their member functions, see the AutoCAD ARX Reference.

Access to AcGi occurs within three member functions of the AcDbEntity base class:

```cpp
virtual Adesk::Boolean AcDbEntity::worldDraw(AcGiWorldDraw *pWd);
virtual void AcDbEntity::viewportDraw(AcGiViewportDraw *pVd);
virtual void AcDbEntity::saveAs(AcGiWorldDraw *pWd,
                                AcDb::SaveType saveType);
```

AutoCAD calls the `worldDraw()` and `viewportDraw()` functions to display the entity. The `saveAs()` function is called by AutoCAD when it needs to cache an alternate graphical representation of the entity for zombie graphics or for the SAVEASR12 command. You must implement `worldDraw()` for any class derived from `AcDbEntity`. The `viewportDraw()` and `saveAs()` functions are optional.

Whenever AutoCAD needs to regenerate the graphics to display an entity, the `worldDraw()` and `viewportDraw()` functions are called in the following manner:

```cpp
if (!entity->worldDraw(pWd))
  for (each relevant viewport)
    entity->viewportDraw(pVd);
```

The purpose of the `worldDraw()` function is to build the portion of the entity’s graphical representation that can be specified independently of any particular model-space view or paper-space viewport context. If any of the entity’s graphics are view-dependent, `worldDraw()` must return `kFalse` and `viewportDraw()` must be implemented. The `viewportDraw()` function then builds the view-dependent portion of the entity’s graphics. Conversely, if the entity has no view-dependent graphics, then `worldDraw()` must return `kTrue` and the custom entity does not implement `viewportDraw()`.

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 **AcGi WorldGeometry** 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 **AcGi SubEntityTraits** 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 (see “Transformations” on page 223):

- **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 methods for querying the viewport’s transformation matrices and viewing parameters.

Overriding AcDbEntity::saveAs()

If your custom entity doesn’t override AcDbEntity::saveAs(), AutoCAD will leverage your worldDraw() function to support SAVEASR12 and zombie entity graphics. (The AcDbEntity::saveAs() function merely calls worldDraw().)

You should override saveAs() if you want to save an alternate graphical representation, either for the SAVEASR12 command or for saving zombie entity graphics, or for both. The saveType parameter (see signature of saveAs() in previous section) is used when you want to build unique, alternate graphical representations for both kinds of saving; it indicates for which purposes saveAs() was called. The saveType parameter has one of the following values:

- kR12Save indicates that saveAs() was called for the SAVEASR12 command
- kR13Save indicates that saveAs() was called to save zombie graphics data

From within saveAs(), you may want to call worldDraw() for one value of saveType and make direct AcGiWorldGeometry and AcGiSubEntityTraits calls for the other value, or you may not want to call worldDraw() at all.

In either case, the saveAs() mechanism works thanks to the magic of object-oriented programming. Before calling saveAs(), AutoCAD first replaces AcGiWorldDraw’s geometry and traits objects with special subclasses of AcGiWorldGeometry and AcGiSubEntityTraits. These subclasses’ 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 of 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 such that its worldDraw() function alone would not produce an appropriate image. In that case, you’ll need to override saveAs() in order to produce reasonable graphics for Release 12 and zombies.
For more information on zombie graphics data, see “Zombies” on page 196.

In the case of the SAVEASR12 command, information about the original R13 entity is not saved in the file. However, the first Release 12 entity will have the same handle as the original Release 13 entity, and any additional Release 12 entities will have the original Release 13 entity handle placed in their xdata under the application name “R13OBJECT.” This feature is provided so that you can group the created Release 12 entities into a block.

Scope of AcGi Objects
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 worldDraw() and viewportDraw(). Once these functions return, the AcGi objects are no longer valid.

Regeneration Type
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.
- kAcGiHideOrShadeCommand performs hidden line removal and indicates that the HIDE or SHADE command is in effect.
- 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. When this regen type is used, only primitives that have area are executed. Polygons, meshes, and shells are rendered, but lines are ignored.

Setting Entity Traits
There are three levels from which entity color, layer, and linetype values can be set: entity level, subentity level, and subprimitive level. Other traits can be set from only one or two of the levels.

- Entity Level: If you don’t set values for color, layer, or linetype, your custom entity will be automatically endowed with the values prescribed by AutoCAD (through the CECOLOR, LAYER, LINE-TYPE, CHANGE, CHPROP commands or by default), exactly as any built-in entity. Additionally, your application can call AcDbEntity member functions such as AcDbEntity::setColorIndex() to set values for these properties at the entity level, but not from within worldDraw(), viewportDraw(), and saveAs(). AcGi does not set trait values at this level.
Subentity Level: You can set values for color, layer, linetype (as well as selection marker and fill type—see next sections) by calling `AcGiSubEntityTraits` functions from your `worldDraw()`, `viewportDraw()`, and `saveAs()` functions. Once a value for the subentity trait is set, it will be used for all subsequently called primitives during the execution of the function unless and until a new value for the same trait is set (except when overridden by subprimitive traits explained below). Trait values set at the subentity level supersede values of the corresponding traits set at the entity level.

Note In this chapter the term subentity is used differently than in chapter 6, “Entities” which uses the term to refer to specific geometric pieces of an entity. In this chapter subentity is not a piece of an entity, but rather 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 211.) 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 flavor 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 entity 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

Color, layer, and linetype are AutoCAD entity properties, so they can also be set at the entity level as described in the previous section. Fill type and GS marker are not AutoCAD entity properties.
Prior to each call to `worldDraw()` and `viewportDraw()`, AutoCAD initializes the color, layer, and linetype subentity traits to their entity-level values. It initializes fill type to correspond to the regen type, and it initializes GS marker to zero (a zero marker signifies “no marker”). Each execution of `worldDraw()` and `viewportDraw()` begins with no subentity traits.

### 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 can’t 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 used in conjunction with the ADS functions `ads_ssget()` and `ads_ssnamex()` permit your application to edit or operate on arbitrary sections of your custom entity objects. “GS Markers and Subentities” in chapter 6, “Entities” describes in detail how to use GS markers (not how to set them), including the use of the `ads_ssget()`, `ads_ssnamex()`, and `AcDbEntity::getSubentPathsAtGsMarker()` functions.
Chapter 10 Deriving from AcDbEntity

The examples in chapter 6 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 contiguously 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::setSelectionMarker()` function, 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:

```c++
// COLOR
static const Adesk::UInt16 kColorByBlock = 0;
staic const Adesk::UInt16 kRed = 1;
staic const Adesk::UInt16 kYellow = 2;
staic const Adesk::UInt16 kGreen = 3;
staic const Adesk::UInt16 kCyan = 4;
staic const Adesk::UInt16 kBlue = 5;
staic const Adesk::UInt16 kMagenta = 6;
stacic const Adesk::UInt16 kWhite = 7;
stacic const Adesk::UInt16 kColorByLayer = 256;

// LINETYPE
static const char* const kNoLinetypeing = "CONTINUOUS";
stacic const char* const kLinetypeByLayer = "BYLAYER";
stacic 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**

The following example illustrates how to get and set entity traits. 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 useful functions, `getLinetypeFromString()` and `getLayerIdFromString()`, which allow you to obtain an ID for a linetype or layer from a string.

```cpp
class Acad : public AcDbDocument
{
    // Function to get linetype ID from string
    static Acad::ErrorStatus getLinetypeIdFromString(const char* str, AcDbObjectId& id);

    // Function to get layer ID from string
    static Acad::ErrorStatus getLayerIdFromString(const char* str, AcDbObjectId& id);

    // Example usage
    Acad::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, here 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 line type 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'd 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 -- 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 = acdbCurDwg()->getLinetypeTable(pLinetypeTable, AcDb::kForRead);
    if (err != Acad::eOk)
        return err;
    // Get the id of the linetype with the name that 'str' contains.
    // "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 -- 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 = acdbCurDwg()->getLayerTable(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 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 proper-
ties. For example, you could use AcGiEdgeData::setColors() to
attach a different color to each edge of the mesh. In the color list,
you 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:
Specifying edge data in a 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. The upper-left face is blue, and the rest of the faces are transparent.

```c++
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. Here, let the rows be cyan and
    // the columns be 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];
    edgeInfo.setColors(pEdgeColorArray);
```
for (i = 0; i < numEdges; i++) {
    pEdgeColorArray[i] =
        i < numRowEdges ? kCyan : kGreen;
}

// 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.setColors(pFaceColorArray);
faceInfo.setVisibility(pFaceVisArray);

for (i = 0; i < numFaces; i++) {
    pFaceVisArray[i] =
        i ? kAcGiVisible : kAcGiInvisible;
    pFaceColorArray[i] = (short)(i + 1);
}

// 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:
Specifying face data in a mesh

Vertex data for the mesh are listed in the same order as in the vertex list. Properties that can be set with this class 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.
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

```cpp
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 = 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. shell() 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:
If the same edge is used in two different faces, properties may conflict. In such cases, you can set one of the edges to 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.

```cpp
Adesk::Boolean
AsdkShellSamp::worldDraw(AcGiWorldDraw* pW)
{
    // Flood 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 2 faces.
    //
    Adesk::UInt32 faceListSize = 8;
    Adesk::Int32 *pFaceList
        = new Adesk::Int32[faceListSize];

    // Face #1
    //
    pFaceList[0] = 3; // 3 vertices in the face
    pFaceList[1] = 0; // pVerts[0]
```
// Face #2
//
pFaceList[4] = 3;  // 3 vertices in the face
pFaceList[5] = 0;  // pVerts[0]

// Apply colors to edges.
//
AcGiEdgeData edgeData;
int numEdges = 6;
short *pEdgeColorArray = new short[numEdges];
edgeData.setColors(pEdgeColorArray);
	pEdgeColorArray[0] = kRed;
pEdgeColorArray[1] = kYellow;
pEdgeColorArray[2] = kGreen;
pEdgeColorArray[3] = kCyan;
pEdgeColorArray[4] = kBlue;
pEdgeColorArray[5] = kMagenta;

// Apply visibility to edges and make common edge
// between two faces have silhouette visibility during
// the HIDE command with ACAD 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];
faceData.setColors(pFaceColorArray);
	pFaceColorArray[0] = kBlue;
pFaceColorArray[1] = kRed;

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 inquired with the following functions:

```cpp
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:

- `kAcGiCounterClockwise`
- `kAcGiNoOrientation`
- `kAcGiClockwise`

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:

```cpp
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 `arcType` 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)
The example in this section shows use of the \texttt{AcGiTextStyle} class. It draws a rectangle around a piece of \texttt{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 \texttt{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:

- \texttt{0x10} Another file (not FONTALT) opened in place of Big-Font file name
- \texttt{0x08} Another file (not FONTALT) opened in place of file name.
- \texttt{0x04} BigFont file name failed to be loaded
- \texttt{0x02} File name failed to be loaded
- \texttt{0x01} Files opened as called for

Text can be scaled in a number of ways. Use \texttt{AcGiTextStyle::setTextSize()} to scale the width and height of the text at the same time. Use \texttt{setXScale()} to scale the width of the text. Use \texttt{setTrackingPercent()} to specify how the characters of a particular font are placed next to each other. If you specify a value of 100, the spacing does not change. If you specify less than 100.0, the characters will squeeze together, and if it's more than 100.0, the characters will be further 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 a "%%%" would not be interpreted as a single percent sign but as three percent signs).

```cpp
Adesk::Boolean AsdkTextStyleSamp::worldDraw(AcGiWorldDraw* pW)
{
    AcGePoint3d pos(0.0, 0.0, 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);
    // Now draw a rectangle around the last text drawn.
    // First you have to create a polyline the size of the
    // bounding box, then you have to transform it to the
    // correct orientation, and then to the location of the
    // text.
```
// Compute the matrix that will orient 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 somewhat.

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;

Converting a Text Style Record into an AcGi Primitive

The next example shows how to convert an AcDbTextStyleRecord
into an AcGiTextStyle primitive.

// Get an AcGiTextStyle from an acDbTextStyleTableRecord.
// Try to map as many characteristics as possible.

void
getTextStyle(AcGiTextStyle& newStyle, AcDbObjectId styleId)
{
  AcDbTextStyleTableRecord *pOldStyle;
  acdbOpenObject((AcDbObject*&)pOldStyle, styleId,
  AcDb::kForRead);
  char *pTmpStr;
  pOldStyle->fileName(pTmpStr);
  newStyle.setFileName(pTmpStr);
  delete pTmpStr;
}
Chapter 10 Deriving from AcDbEntit

```cpp
cpOldStyle->bigFontFileName(pTmpStr);
newStyle.setBigFontFileName(pTmpStr);
delete pTmpStr;

newStyle.setTextSize(pOldStyle->textSize());
newStyle.setXScale(pOldStyle->xScale());
newStyle.setObliquingAngle(pOldStyle->obliquingAngle());

pOldStyle->close();
newStyle.loadStyleRec();
```

## 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 in order 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:

![tessellation_diagram](image)

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

```cpp
virtual double AcGiWorldDraw::deviation(AcGiDeviationType,
const AcGePoint3d&) const = 0;
```

The deviation types are

- `kAcGiMaxDevForCircle` (for circles and arcs)
- `kAcGiMaxDevForCurve`
- `kAcGiMaxDevForBoundary`
Displaying the Entity

- `kAcGiMaxDevForIsoline`
- `kAcGiMaxDevForFacet` (for surfaces; formula for calculating this deviation uses the value of the FACETRES system variable)

You can bypass AutoCAD's attempt to optimize tessellation for different types of geometry by using the circle deviation (`kAcGiMaxDevForCircle`) for any entity.

**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.

```cpp
virtual Adesk::UInt32 AcGiWorldDraw::numberOfIsolines() const = 0;
```

**Viewports**

A viewport provides you with a vantage point on world space. It is defined by a camera that is pointed at world space, with a certain field of view. Objects to the left and right of the field of view, as well as those in front and behind it, are not visible on the screen. In AutoCAD, the field of view is determined by the ZOOM factor if perspective is off, or by the DVIEW command if perspective is on.

The `AcGiViewport` class provides methods for obtaining information about the viewpoint, such as `getCameraLocation()` and `getCameraUpVector()`, as well as whether you are in perspective mode (`getPerspective()`). It also provides methods for obtaining the various transformations (`getModelToEyeTransform()`, `getEyeToModelTransform()` and perform clipping).

**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 perspective is off, eye and display coordinates are identical.
In AutoCAD, when the RENDER command is given, the entity's world coordinates are sent to a special graphics pipeline that determines how to display the entity on the screen.

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 AcGiViewport class provides methods 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() methods, 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.
This section contains four examples. The first example shows drawing 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. It shows use of `isPerspective()`, `doPerspective()`, `getFrontAndBackClipValues()`, `polylineDc()`, `polylineEye()`, and `polyline()`.

Converting the eye-coordinate line segment to display space requires two steps:

1. If a view has clipping planes in force, dip the eye-coordinate line segment to them.
2. If perspective is on, then perform the conversion from eye coordinates to perspective.

The example uses pseudocode for the clipping operation.

```c
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 its 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->subEntT##traits().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 display space equivalent of the
// model space polygon.
//
Pv->subEntT##traits().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.

If you’re using the polygonEye(), polygonDc(), polylineEye(), or polylineDc() methods of AcGiViewportGeometry, you need to 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);
}
Displaying the Entity

Acad::ErrorStatus
AsdkViewGeomSamp::transformBy(const AcGeMatrix3d &xfm)
{
    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 ACAD DISPLAY mode,
    //
    if (pW->regenType() == kAcGiStandardDisplay)
    {
        // from each viewport's vantage point, figure out
        // which sides of the pyramid would be visible,
        // then draw 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, or RENDER, 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 the viewport and which are not.
    int 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 or toward the screen.)
// Draw the 6 edges connecting the vertices using eye
// coordinate geometry that can be back and front
// clipped.

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 use of AcGiViewportDraw::polylineDc() and AcGiViewport::getViewportDcCorners() to obtain the display coordinates of the viewport. This method 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 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.

```cpp
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;
}
```
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 gives you 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.

![Diagram of circle and angle α](image)

Adesk::Boolean
AsdkTesselateSamp::worldDraw(AcGiWorldDraw *pW)
{
  // draw a red 1 x 1 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);
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) all 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 is a problem of
        // figuring out the least number of segments needed
        // by a polyline so it doesn't look segmented.
    }
}
// By the way, 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 very high resolution image internally
// that 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 upon.)

double radius = 0.5;
double half_pixel_hgt = 2.0 / area.x; // in world
// coords
int num_segs = 8;
double angle = two_pi / num_segs;

if (half_pixel_hgt > radius / 2) {
    // The circle is around 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 otherwise 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;
}

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 use object snap points, grip points, and stretch points, as well as how to override the entity transformation functions.

Object Snap Points

You’ll need to override the `getOsnapPoints()` function if you want your custom entity to support object snap modes. AutoCAD invokes this method 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:

```cpp
AcDb::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)
        return Acad::eOk;

    int startIndex = gsSelectionMark - 1;
    AcGePoint3dArray vertexArray;
    if ((es = getVertices(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:
        break;
    case AcDb::kOsModeNear:
        break;
    case AcDb::kOsModePerp:
        break;
    case AcDb::kOsModeCen:
        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:
    vec = vertexArray[startIndex + 1] - vertexArray[startIndex];
    vec.normalize();
    line.set(vertexArray[startIndex], vec);
    pt = line.projClosestPointTo(pickPoint, viewDir);
    snapPoints.append(pt);
    break;

case AcDb::kOsModePerp:
    // Create semi-infinit 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(mCenter);
    break;

default:
    return Acad::eOk;
}
return es;

Grip Points
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 getGripPoints() and moveGripPointsAt() 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 getGripPoints() are
not currently used.

Stretch mode in grip editing allows you to stretch an object by moving
selected grips to new locations. AutoCAD calls moveGripPointsAt() 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, moveGripPointsAt() call
transformBy().

Note The default implementation of AcDbEntity::moveGripPointsAt() is to invoke transformBy().

When the user is in Grip Move, Rotate, Scale, or Mirror modes,
AutoCAD calls the transformBy() method, 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
moveGripPointsAt(), 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 trans-
lated 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 = this->getVertices(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);
return es;
}

Acad::ErrorStatus
AsdkPoly::moveGripPointsAt(
    const AcDbIntArray& indices,
    const AcGeVector3d& offset)
{
    assertWriteEnabled();
    Acad::ErrorStatus es = Acad::eOk;
    AcDbIntArray move;

    int numGripPoints = mNumSides + 1;

    move.setLogicalLength(numGripPoints);
    move.setAll(Adesk::kFalse);
    for (int j = 0; j < indices.length(); j++) {
        if (indices[j] < numGripPoints) {
            move[indices[j]] = Adesk::kTrue;
        }
    }

    for (int i = 0; i < numGripPoints; i++) {
        if (move[i]) {

            // If one of the vertices of the polygon is
            // moved, find out the new start point from that
            // movement and set it.
            //
            // One way to do that would be to define a new
            // polygon taking the moved point as the start
            // point. Get the vertices of that polygon, and
            // the start point of the old polygon would be
            // one of the vertices of the new polygon at the
            // appropriate index.
            //
            // If the center is moved, just translate the
            // polygon.
            //
            if (i < mNumSides) { // Vertex movement.
                AcGePoint3dArray oldVertexArray;
                AOK(this->getVertices(oldVertexArray));

                AcGePoint3d vertex = oldVertexArray[i];
                vertex += offset;
                if (vertex == mCenter) {
                    return Acad::eOk;
                }
            }
        }
    }
}
AsdkPoly* pNewPoly = new AsdkPoly(mCenter, vertex, mNumSides, mPlaneNormal, mpName);
AcGePoint3dArray newVertexArray;
AOK(pNewPoly->getVertices(newVertexArray));
delete pNewPoly;
mStartPoint = newVertexArray[mNumSides - i];
} else { // Center movement.
    AcGeMatrix3d mat;
    mat.entry[0][3] = offset[X];
    mat.entry[1][3] = offset[Y];
    mat.entry[2][3] = offset[Z];
    this->transformBy(mat);
}
return es;

Stretch Points
The set of stretch points for an entity is often a subset of its grip points. When the user invokes the STRETCH command, getStretchPoints() 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 AcDbEntity::getStretchPoints() and AcDbEntity::moveStretchPointsAt() is to invoke your getGripPoints() and moveGripPointsAt().

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 getGripPoints() and transformBy().
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 `transformBy()`. Otherwise, it invokes `moveGripPointsAt()`.

```cpp
Acad::ErrorStatus
AsdkPoly::getStretchPoints(
    AcGePoint3dArray& stretchPoints) const
{
    assertReadEnabled();
    Acad::ErrorStatus es;
    if ((es = this->getVertices(stretchPoints))
        != Acad::eOk)
    {
        return es;
    }
    // Remove the duplicate point at the start/end.
    stretchPoints.removeAt(stretchPoints.length() - 1);
    return es;
}
```

```cpp
Acad::ErrorStatus
AsdkPoly::moveStretchPointsAt(
    const AcDbIntArray& indices,
    const AcGeVector3d& offset)
{
    assertWriteEnabled();
    Acad::ErrorStatus es = Acad::eOk;
    if (indices.length() == 0 || offset.isZeroLength()) {
        return Acad::eOk;
    }
    // If all the stretch points are selected call
    // transformBy(), otherwise call moveGripPoints().
    // if (indices.length() >= mNumSides) {
    AcGeMatrix3d mat;
    mat[0, 3] = offset[0];
    mat[1, 3] = offset[1];
    mat[2, 3] = offset[2];
    return transformBy(mat);
    } else {
        return moveGripPointsAt(indices, offset);
    }
}
Common Entity Functions

Transformation Functions

The AcDbEntity class offers two transformation functions. The first, transformBy(), applies a matrix to an entity. The second, getTransformedCopy(), 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 AcDbEntity::getTransformedCopy() clones the entity and then invokes transformBy() on the cloned entity. (Use AcGeMatrix3d::isUniScaledOrtho() to determine if the input matrix is uniformly scaled and orthogonal.)

The custom AsdkPoly class overrides both transformBy() and getTransformedCopy(). When AsdkPoly is nonuniformly scaled, it becomes a polyline.

```cpp
Acad::ErrorStatus AsdkPoly::transformBy(const AcGeMatrix3d& xform)
{
    assertWriteEnabled();
    mCenter.transformBy(xform);
    mStartPoint.transformBy(xform);
    mPlaneNormal.transformBy(xform);
    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 = this->getVertices(vertexArray)) != Acad::eOk)
    {
        return es;
    }
    for (int i = 0; i < vertexArray.length(); i++) {
        vertexArray[i].transformBy(mat);
    }
    AcDb3dPolyline* pPline = NULL;
    if ((es = rx_makePline(vertexArray, pPline)) != Acad::eOk)
    {
        return es;
    }
    assert(pPline != NULL);
    pPline->setPropertiesFrom(this);
    return es;
}
```
Intersecting with Other Entities

When you implement `intersectWith()` for your custom entity, be aware that this function is not implemented for all Release 12 entities. For those entities, you’ll need to use the AcGe library code to test for intersection.

The `intersectWith()` method has two forms:

```cpp
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 `intersectWith()` tests for simple intersection of two entities. The second form calculates the intersection on a projection plane. However, both methods return the intersection points on the entity itself.

If you use the projection plane form of `intersectWith()`, then you’ll need to perform the following steps:

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.

```cpp
Acad::ErrorStatus
AsdkPoly::intersectWith(
    const AcDbEntity* ent,
    Acad::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, i.e. the ones that have their
    // 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;
    }
```

Chapter 10 Deriving from AcDbEntit

```cpp
} else {
    AcGePoint3dArray vertexArray;
    if (Acad::eOk != this->getVertices(vertexArray))
        return es;
    if (intType == kExtendArg
        || intType == kExtendBoth)
        intType = 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(this->normal());
        if (Acad::eOk != ent->intersectWith(pAcadLine, intType,
                                             points))
            delete pAcadLine;
            return es;
        delete pAcadLine;
    }
    return es;
}
```

ErrorStatus AskPoly::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, i.e. the ones that have their
    // 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 = this->getVertices(vertexArray))
        != Acad::eOk)
    {
        return es;
    }
    if (intType == kExtendArg
        || intType == kExtendBoth)
    {
        intType = kExtendThis;
    }
    AcDbLine* acadLine;
    for (int i = 0; i < vertexArray.length() - 1; i++) {
        acadLine = new AcDbLine();
        acadLine->setStartPoint(vertexArray[i]);
        acadLine->setEndPoint(vertexArray[i + 1]);
        acadLine->setNormal(this->normal());
        if ((es = ent->intersectWith(acadLine, intType,
            &projPlane, points)) != Acad::eOk)
        {
            delete acadLine;
            return es;
        }
        delete acadLine;
    }
}
All the points that we got in this process are on
the other curve and 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.

So 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.

```cpp
AcGePoint3d projPt, planePt;
AcGePoint3dArray pts;
AcGeLine3d line;
AcGePlane polyPlane(mCenter, mPlaneNormal);
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

ARX 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 where the user selects your custom entity in an operation that involves it intersecting with another custom entity that you are not aware of. The following guidelines should help you implement the intersectWith() method 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() method 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() method) to a set of recognizable native entities, then turn around and call intersectWith() on the entity that came in as argument to your intersectWith() method. 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() method 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 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.
Using AcEdJig

The AcEdJig class is used to perform drag sequences, usually to acquire a new entity—to create it, edit it, and add it to the database. If you are deriving a new entity class, you will often want to implement your own version of AcEdJig as well. 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 dragger. 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. Also do not use this class with AutoCAD Release 12 entities.

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, supplying graphical feedback specified by a cursor type and a single entity. To use this class, follow these basic steps (described in detail in later sections):

1. Create an instance of your derived class of AcEdJig.
2. If you are using a prompt with keywords, invoke AcEdJig::setKeywordList().
3 Establish your prompt text with `AcEdJig::setDispPrompt()`.

4 If you want to set a special cursor type, call `AcEdJig::setSpecialCursorType()`. (This step is optional and can typically be omitted.)

5 If desired, place limitations on the drag sequence and the return value using the `AcEdJig::setUserInputControls()` function.

6 Call `AcEdJig::drag()`, which controls the drag loop and in turn calls the `sampler()`, `update()`, and `entity()` functions until the user ends the drag sequence.

7 Check the return status from `AcEdJig::drag()` and commit the changes of the drag sequence. If the user canceled or aborted the process, perform appropriate cleanup.

### Setting Up Parameters for the Drag Sequence

Before you call the `AcEdJig::drag()` function, you need to set up certain parameters for the drag sequence: the keyword list, display prompt, 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:

```cpp
void AcEdJig::setKeywordList(const char* kyWdList);
```

The keyword list is a single string with each keyword separated 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:

```cpp
void AcEdJig::setDispPrompt(const char* prompt);
```
### Cursor Types

If you want to set a special cursor type, use the following function:

```cpp
void AcEdJig::setSpecialCursorType(AcEdJig::CursorType);
```

The `CursorType` can be one of the values in the following table:

<table>
<thead>
<tr>
<th>Cursor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kCrosshair</td>
<td>Crosshairs aligned with the User Coordinate System.</td>
</tr>
<tr>
<td>kRectCursor</td>
<td>Rectangular window cursor aligned with the display coordinate system.</td>
</tr>
<tr>
<td>kRubberBand</td>
<td>Same as kCrosshair, except also displays a rubber band from the base point.</td>
</tr>
<tr>
<td>kTargetBox</td>
<td>OSNAP cursor; similar to kEntitySelect cursor, except its size is controlled by the system variable $APERTURE.</td>
</tr>
<tr>
<td>kCrosshairNoRotate</td>
<td>Crosshairs aligned with the display coordinate system.</td>
</tr>
<tr>
<td>kInvisible</td>
<td>No cursor graphics; only entity graphics are displayed.</td>
</tr>
<tr>
<td>kEntitySelect</td>
<td>Single entity pick box; the entity is not actually selected in this case. Entity selection is handled with ads_ssget().</td>
</tr>
<tr>
<td>kParallelogram</td>
<td>Rectangle aligned with the User Coordinate System (can be a parallelogram on the display).</td>
</tr>
<tr>
<td>kEntitySelectNoPersp</td>
<td>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.</td>
</tr>
<tr>
<td>kPكشفOrGrips</td>
<td>Default cursor; what the cursor looks like &quot;between&quot; commands.</td>
</tr>
<tr>
<td>kArrow</td>
<td>Displays the arrow cursor used for dialog boxes in AutoCAD</td>
</tr>
</tbody>
</table>
This step is optional. The `acquirePoint()` methods 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:

```cpp
void set_UserInputControls(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 `UseInputControls` can be one of the following values:

- `kGoverned by OrthoMode`
- `kNullResponseAccepted`
- `kDontEchoCancelForCtrlC`
- `kDontUpdateLastPoint`
- `kNoDwgLimitsChecking`
- `kNoZeroResponseAccepted`
- `kNoNegativeResponseAccepted`
- `kAccept 3dCoordinates`
- `kAcceptMouseUpAsPoint`
- `kAnyBlankTerminatesInput`
- `kInitialBlankTerminatesInput`
Drag Loop

After you have set up the parameters for the drag sequence, you call `AcEdJig::drag()`, which performs the drag loop until the user presses 5 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 `AcEdJig::sampler()`. The `sampler()` function calls one of the `acquireXXX()` functions to obtain a geometric value (an angle, distance, or a point). The function always returns immediately after polling the current pointing device position.
3. If there is no change in the geometric value sampled, your `sampler()` function can return `kNoChange` and return to step 1.
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 `AcEdJig::update()`, using the acquired geometric value to update the entity.
6. The dragger then calls `AcEdJig::entity()`, passing in a pointer to the entity to be regenerated. Next, the dragger calls `worldDraw()` on the entity to regenerate it.
7. Return to step 1 unless the current dragger event was generated by the user picking with the pointing device, pressing Cancel, or issuing a string termination character to end dragging.

The following flowchart shows these steps.
```
using AcEdJig

get an event

<sampling>
rs=AcEdJig::acquireXXX()

rs==kNoChange

Yes

return rs

No

Do you need to update the drag image?

No

Yes

return rs other than kNoChange

update

entity->worldDraw

event was a digitizer pick of string terminator

- Return from AcEdJig::drag()
```
Implementing the sampler(), update(), and entity() Functions

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 necessary further analysis on the obtained geometric value and drag status. You’ll 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. (Or, 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 it's 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;
};

// 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), mAxisPt(),
        mMajorAxis(), 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;
    setUserInputControls((UserInputControls)
        (kGovernedByOrthoMode | kAccept3dCoordinates));
// 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 in order to
// acquire a sample input
// AcEdJig::DragStatus
AsdkEllipseJig::sampler()
{
    DragStatus stat;
    if (mPromptCounter == 0) {
        // Acquire the major axis endpoint
        stat = acquirePoint(mAxisPt, mCenterPt);
    } else if (mPromptCounter == 1) {
        // Acquire the distance from ellipse center to minor
        // axis endpoint this will be used to calculate the
        // radius ratio.
        stat = acquireDist(mRadiusRatio, mCenterPt);
    }
    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 (i.e. 1/2 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
    // 1/2 of the actual minor axis length.  Since
    // AcDbEllipse stores the major axis vector as the
    // vector from the center point to the ellipse curve
    // (1/2 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 then
// drags to visually select the major and minor axes
// lengths.  This sample is somewhat limited in that 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;

    ads_getpoint(NULL, "Ellipse center point: ",
                  asDblArray(tempPt));
Chapter 10 Deriving from AcDbEntit

// The point we just got is in UCS coords, 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

ads_trans(asDblArray(tempPt), &rbFrom, &rbTo, Adesk::kFalse, asDblArray(tempPt));

// Now need to get the current UCS Z-Axis to be used as the normal vector for the ellipse.
//
AcGeVector3d x = acdbCurDwg()->ucsXdir();
AcGeVector3d y = acdbCurDwg()->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();

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* pkt)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(pkt);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
        break;
    }
    return AcRx::kRetOK;
}
PART III

SPECIALIZED TOPICS

CHAPTER 11 ■ NOTIFICATION

CHAPTER 12 ■ TRANSACTION MANAGEMENT

CHAPTER 13 ■ DEEP CLONING

CHAPTER 14 ■ PROTOCOL EXTENSION

CHAPTER 15 ■ USING MFC WITH AN ARX APPLICATION
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.
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 your application, follow these general steps:

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 you are finished using the reactor, follow these steps:

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 8, “Deriving a custom ARX Class” and chapter 9, “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**
- AcRxDLinkerReactor
- AcEditorReactor
- AcDbDatabaseReactor
- AcTransactionReactor
- AcDbObjectReactor
- AcDbEntityReactor

**Types of Object Reactors**

The reactor classes shown in boldface above are also referred to as transient reactor classes. If you want your program to receive event notification, you'll usually use a transient reactor, 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, involves use of 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.

The mechanism for using an AcDbObject as a reactor is analogous to that for using a transient reactor class:

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, preferably to 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 take care of deleting the object reactor because it is a database object.
To use a transient reactor, derive a new class from one of the following base classes:

- **AcRxDLinkerReactor** (monitors ARX 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, 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, you need to use only standard C++ techniques for creating new transient reactor classes. The ARX 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()**
- **modifyUndone()**
- **modifiedXData**
- **unappended()**
- **reappended()**
- **objectClosed()**
Using Reactors

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). The adding or deleting of a transient reactor is not monitored by the undo mechanism. (For persistent reactors, the notifier object has to be opened for write and the addition or removal of the reactor 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 method of `AcDbObject`. In the override of this method 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 methods 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()`, `saveComplete()`. Although ADS library calls are permitted within reactor notification functions, no LISP or ADS application interaction can be performed within the notification function. For example, don't use calls such as `ads_getpoint()` or `ads_grread()`.

See chapter 13, “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.

```cpp
class AsdkDbReactor {
  long gEntAcc = 0; // Global entity count
  AsdkDbReactor *gpDbr = NULL; // pointer to database reactor

  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");
    ads_printf(" Db==%lx\n", (long) db);
    gEntAcc++;
    ads_printf("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");
    ads_printf(" 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++;
    }
    ads_printf(" Db==%lx\n", (long) db);
    ads_printf("Entity Count = %d\n", gEntAcc);
}

// Prints the message passed in by eventStr and then
// proceeds to call printObj to print the information about
// the object that's triggered the notification.
// void printDbEvent(const AcDbObject* pObj, const char* pEvent) {
    ads_printf(" 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) {
        ads_printf("(NULL)\n");
        return;
    }

    AcDbHandle objHand;
    char handbuf[17];

    // Get the handle as a string
    pObj->getAcDbHandle(objHand);
    objHand.getIntoAsciiBuffer(handbuf);
    ads_printf("\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
// effects because subsequent calls will be ignored.
//
void
watchDb()
{
    if (gpDbr == NULL) {
        gpDbr = new AsdkDbReactor();
    }
    acdbCurDwg()->addReactor(gpDbr);
    ads_printf("  Added Database Reactor to acdbCurDwg().\n\n");
}

// Removes the database reactor
//
void
clearReactors()
{
    if (acdbCurDwg() != NULL) {
        acdbCurDwg()->removeReactor(gpDbr);
        delete gpDbr;
        gpDbr = NULL;
    }
}
// Arx entry point function
// AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(pkt);
        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;
    case AcRx::kLoadDwgMsg:
        // New AcDbDatabase, so if reactor exists
        // attach it.
        //
        if (gpDbr != NULL) {
            watchDb();
        }
    }
    return AcRx::kRetOK;
}

Using an Object Reactor

If you want one database object to react to another database object, you can derive a class from AcDbObject (or any of its subclasses), implement the notification functions for it, and then 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 ARX 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 ARX 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` method. If it is a persistent reactor, you can use the appropriate method to obtain its object ID. If it is not a persistent reactor, you can cast the entry to `AcDbObjectReactor`.

```c++
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);
            ads_printf("Persistent reactor found.");
            // Let’s echo the keyname to the user...
            //
            char *keyname;
            getPersReactorKey(keyname, persObjId);
            ads_printf("This is the reactor named %s", keyname);
            // Let’s open it up and see if it’s one of
            // ours. If it is, let’s fire the custom
            // notification.
            //
            if ((retStat =
                acdbOpenAcDbObject(pPersReacObj,
                persObjId, AcDb::kForNotify))
                != Acad::eOk)
                {
                ads_printf("\nFailure for\n" 
                "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));
    ads_printf("\n
Transient reactor found");
    // Let's just report what kind we found...
    //
    if (pObjReactor->isKindOf(
            AsdkSimpleObjReactor::desc()))
    {
        ads_printf(" of type\n        AsdkSimpleObjReactor");
    } else if (pObjReactor->isKindOf(
            AcDbEntityReactor::desc()))
    {
        ads_printf(" of type\n        AcDbEntityReactor");
    } else if (pObjReactor->isKindOf(
            AcDbObjectReactor::desc()))
    {
        ads_printf(" of type\n        AcDbObjectReactor");
    } else {
        ads_printf(" of unknown type.");
    }
}
} else {
    ads_printf("\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, 0, \n    AsdkObjectToNotify, persreac);
    // This function is called everytime 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();
            ads_printf("This is a %s.\n", cstr);
            ads_printf("I only work with lines. Sorry.\n");
            return;
        }
        ads_printf("Reactor attached to %lx calling %lx.\n", pLine->objectId(), mId);
    }
Using Reactors

// This open will fail during notification caused by
// 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 cross 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 object's information in from DWG, UNDO, etc.
//
// Acad::ErrorStatus AsdkObjectToNotify::dwgInFields(AcDbDwgFiler* filer)
//{
//    assertWriteEnabled();
//    AcDbObject::dwgInFields(filer);
//    filer->readItem(&mFactor);
//    filer->readItem((AcDbSoftPointerId*) &mId);
//    return filer->filerStatus();
//}

// Files object's information out to DWG, UNDO, etc.
//
// Acad::ErrorStatus AsdkObjectToNotify::dwgOutFields(AcDbDwgFiler* filer) const
//{
//    assertReadEnabled();
//    AcDbObject::dwgOutFields(filer);
//    filer->writeItem(mFactor);
//    filer->writeItem((AcDbSoftPointerId&)mId);
//    return filer->filerStatus();
//}
// Files object's information in from DXF and LISP/ADS
// 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->atSubclassList("AsdkObjectToNotify"))
    //    return Acad::eBadDxfSequence;
    struct resbuf rbIn;
    while (es == Acad::eOk) {
        if ((es = filer->readItem(&rbIn)) == Acad::eOk) {
            if (rbIn.retype == AcDb::kDxfReal)
                mFactor = rbIn.resval.rreal;
            else if (rbIn.retype == AcDb::kDxfSoftPointerId)
                // ObjectIds are filed in as ads_names
                // acdbGetObjectId(mId, rbIn.resval.rname);
            else  // invalid group
                return(filer->pushBackItem());
        }
    }
    return filer->filerStatus();
}

// Files object's information out to DXF and LISP/ADS
// 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 = acdbCurDwg();
    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);
    ads_printf("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);
    ads_printf("Line B is %lx from 1,2 to 2,2.\n", pLineB->objectId());

    // Open the named object dictionary, 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 lineA

    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;
    ads_printf("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 LineB
//
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;
    ads_printf("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;
    acdbCurDwg()->getBlockTable(pBlockTable, AcDb::kForRead);
    pBlockTable->getAt(ACDB_MODEL_SPACE, pSpaceRecord, AcDb::kForWrite);
    pBlockTable->close();
    pSpaceRecord->appendAcDbEntity(objId, pEntity);
    pSpaceRecord->close();
    return;
}
// Initialization function called from acrxEntryPoint during kInitAppMsg case. This function is used to add commands to the command stack.
//
// void initApp()
//
//    aecedRegCmds->addCommand("ASDK_ALINES", "ASDK_ALINES", "ALINES", ACRX_CMD_MODAL, assocLines);
//    AsdkObjectToNotify::rxInit();
//    acrxBuildClassHierarchy();
//
// Clean up function called from acrxEntryPoint during the kUnloadAppMsg case. This function removes this apps command set from the command stack.
//
// void unloadApp()
//
//    aecedRegCmds->removeGroup("ASDK_ALINES");
//    AsdkObjectToNotify::rxUninit();
//    acrxBuildClassHierarchy();
//    deleteAcRxClass(AsdkObjectToNotify::desc());

// ARX entry point
//
// AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
//
//    switch (msg) {
//        case AcRx::kInitAppMsg:
//            acrxDynamicLinker->unlockApplication(pkt);
//            initApp();
//            break;
//        case AcRx::kUnloadAppMsg:
//            unloadApp();
//            break;
//    }
//    return AcRx::kRetOK;
Immediate Versus Commit-Time Events

For AcDbObjectReactor, notification events can be either immediate or 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 method 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)

Triggering of immediate notifications occurs 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()`
- `erased()`
- `modifyUndone()`
- `modifiedXData()`
- `unappended()`
- `reappended()`
- `graphicsModified()`

As an example of commit-time notification, consider the `modified()` notification of AcDbObjectReactor. Suppose an object is opened and a modification method is called on it. The modification method calls `assertWriteEnabled()` and all reactors receive the `openedForModify()` reaction. Subsequent modification methods 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 results in an abort of AutoCAD with the error message `eWasNotOpenForWrite` or `eInProcessOfCommitting`.

You can use any of 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()` method 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, try to adhere to the following guidelines. Overuse of notifications that violate these guidelines could result in unpredictable results for your application if the internal implementation of notification changes.

- Do not rely on the sequence of notification firing.

  It's recommended that you do not rely on the sequence of notification firing. Examples of sequences that you can count on being maintained in the future are commandWillStart() being fired before commandEnded(), and beginInsert() before endInsert(). Relying on more complex sequences may result in problems for your application if the sequence is changed as a result of new notifications being introduced in the future and existing ones being rearranged.

- Do not rely on the sequence of operations (function calls) between notifications.

  It is not guaranteed that certain methods will be called between certain notifications. If you tie your application to this level of detail, there is a very high probability of your application breaking in future releases.

  Instead of relying on sequences, rely on notifications to indicate the state of the system. For example, when you receive erased(kTrue) notification on object A, all it means is that object A is erased. If you receive erased() notification on A followed by an erased() notification on B, all it means is that both objects A and B are erased. It doesn't mean that the system will guarantee that B will always be erased after A.

- Do not use any ADS functions in your notification callback function (for example, do not use ads_command(), ads_getpoint(), ads_getkword(), or any other ads_ function).

  Similar interpretations apply to notifications on database reactors, editor reactors, and transaction reactors.
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.”
Overview

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 operation 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 be incurring 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 is 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

```c
#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 “Nested 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 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 a 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 you are returned a pointer to it (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 transaction that contains this 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, then end or abort the transaction before you return from that function because while in the function you still have the knowledge of a transaction that you’re responsible for. 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 `ads::command()` or a transparent command ends, but they should all be resolved when a main command ends and AutoCAD returns to 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
- NEW
- QUIT
- SAVE
- UNDO
- DXFIN
- OPEN
- RECOVER
- SCRIPT
- XREF
- INSERT
- PURGE
- REDO
- U

**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()` methods take an *open mode* as an argument 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()` method 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.”

Object pointers obtained through `getObject()` are dealt with at the time of end or abort transaction and should not be used for calling `close()`.

**Newly Created Objects and Transactions**

You will probably instantiate new objects in your application and add them to the database. There are two ways you can 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()` method described previously to obtain a brand new pointer for your operations. Even if you call `close()` on the object after adding it to the database, its creation will get undone if the containing transaction is aborted. See the subsequent section “Mixing the Transaction Model with the Open and Close Mechanism.”

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::addNewlyCreatedDBObject()` or `AcTransaction::addNewlyCreatedDBObject`. 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 288) and begins the commit process where 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 \texttt{AcTransactionManager::queueForGraphicsFlush()} and \texttt{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 are not committed to the database. \texttt{AcTransactionManager::queueForGraphicsFlush()} queues up all the eligible entities associated with all the transactions for graphics update and \texttt{AcTransactionManager::flushGraphics()} draws them. You can also use \texttt{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 \texttt{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:

\begin{verbatim}
virtual void transactionStarted (int& numTransactions);
virtual void transactionEnded   (int& numTransactions);
virtual void transactionAborted (int& numTransactions);
virtual void endCalledOnOutermostTransaction (int& numTransactions);
\end{verbatim}

The first three notifications are fired when any transaction, including the nested ones, is started, ended, or aborted. You can use these notifications in conjunction with \texttt{AcTransactionManager::numActiveTransactions()} to determine the transaction that is relevant to the notification. For example, if a call to \texttt{AcTransactionManager::numActiveTransactions()} returns zero in your override of \texttt{AcTransactionReactor::transactionEnded()} or \texttt{AcTransactionReactor::transactionAborted()}, you know the outermost transaction is ending or aborting.

The \texttt{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: Nested Transactions

The following example includes three nested transactions. The following is the sequence of events:

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 that 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.


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:

```c
void transactCommand()
{
    Acad::ErrorStatus es = Acad::eOk;
    abortCount++;

    // Create a poly and post it to the database.
    // ads_printf("Creating a poly...Please supply the";
    // " required input.");
    if ((es = createAndPostPoly()) != Acad::eOk)
        return;
```
// Start a transaction
//
AcTransaction *pTrans
    = actrTransactionManager->startTransaction();
assert(pTrans != NULL);
ads_printf("\n\n#### Started transaction one.\n" "####\n\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 (ads_entsel("\nSelect a polygon: ",
                      ename, pickpt))
    {
    case RTNORM:
        acdbGetObjectId(objId, ename);
        if (! (es = pTrans->getObject(pObj, objId,
                                       AcDb::kForRead)) )
            return;
        assert(pObj != NULL);
pPoly = AsdkPoly::cast(pObj);
        if (pPoly == NULL) {
            ads_printf("\nNot a polygon. Try again\n");
            continue;
        }
        break;
    case RTNONE:
    case RTCAN:
        actrTransactionManager->abortTransaction();
        return;
    default:
        continue;
    }
}
Now that we have a poly, convert it to a region and extrude it.

// We will be extruding the poly.
ads_printf("\n\n\nWe will be extruding the poly.");
AcGePoint3d center = pPoly->center();
ads_point pt;
double height;
if (ads_getdist(pt, "Enter Extrusion height: ",
    &height) != RTNORM)
{
    actrTransactionManager->abortTransaction();
    return;
}
if ((es = extrudePoly(pPoly, height)) != 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;
pPoly->getCoordSys(mat);
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->addNewlyCreatedDBObject(pSolid);
pSolid->draw();
ads_printf("\nCreated a cylinder at the center of the poly.\n\n");
// 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);
ads_printf("\n\n##### Started transaction two.#####\n");

AcDb3dSolid *pExtrusion, *pCylinder;
if ((es = getASolid("Select the extrsion: ", pTrans,
   AcDb::kForWrite, savedExtrusionId, pExtrusion))
!= Acad::eOk)
{
   actrTransactionManager->abortTransaction();
   actrTransactionManager->abortTransaction();
   return;
}
assert(pExtrusion != NULL);
if ((es = getASolid("Select the cylinder: ", pTrans,
   AcDb::kForWrite, savedCylinderId, pCylinder))
!= Acad::eOk)
{
   actrTransactionManager->abortTransaction();
   actrTransactionManager->abortTransaction();
   return;
}
assert(pCylinder != NULL);
pExtrusion->booleanOper(AcDb::kBoolSubtract, pCylinder);
pExtrusion->draw();
ads_printf("\nSubtracted the cylinder from the extrusion.\n");
// At this point, cylinder is a NULL solid. We might
// as well erase it.
//
assert(pCylinder->isNull());
pCylinder->erase();

// Start another transaction and slice the resulting
// solid into two halves.
//
// pTrans = actrTransactionManager->startTransaction();
assert(pTrans != NULL);
ads_printf("\n\n##### Started transaction three.#####\n");

AcGeVector3d vec, normal;
vec = pPoly->startPoint() - pPoly->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);
float mat, vec[0], vec[1], vec[2];
pOtherHalf->transformBy(mat);
AcdDbObjectId otherHalfId;
addToDb(pOtherHalf, otherHalfId);
actrTransactionManager
  ->addNewlyCreatedDBRObject(pOtherHalf);
pOtherHalf->draw();
pExtrusion->draw();
ads_printf("Sliced the resulting solid into half"
            " and moved one piece.\n");

// After all this work, let's abort transaction three,
// so that we are back to the hole in the extrusion.
//
Adesk::Boolean yes = Adesk::kTrue;
if (getYOrN("Let's abort transaction three, yes?" 
            "[Y] : ", Adesk::kTrue, yes) == Acad::eOk 
            && yes == Adesk::kTrue)
{
  ads_printf("Aborting transaction three.\n");
  actrTransactionManager->abortTransaction();
  abortCount++;
  ads_printf("Back to the un-sliced solid.\n");
  pExtrusion->draw();
  char option[256];
  ads_getkword("Hit any key to continue.", option);
} else {
  ads_printf("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's the slice
// we really wanted.
//
pTrans = actrTransactionManager->startTransaction();
assert(pTrans != NULL);
ads_printf("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);
actrTransactionManager->addNewlyCreatedDBRObject(pOtherHalf);
pOtherHalf->draw();
pExtrusion->draw();
ads_printf("\nSliced the resulting solid into half\n" "along a plane");
ads_printf("\nPerpendicular to the old one and moved\n" "one piece.");

// Now, optionally, let's end all the transactions.
//
yes = Adesk::kFalse;
if (getYOrN("\nAbort transaction three? [N] : ",
Adesk::kFalse, yes) == Acad::eOk
&& yes == Adesk::kTrue)
{
  ads_printf("\n\n$$$$$$  Aborting transaction\n" "three. $$$$$$");
  actrTransactionManager->abortTransaction();
  ads_printf("\nBack to the un-sliced solid.");
} else {
  ads_printf("\n\n>>>>>>  Ending transaction three.\n" "<<<<<<\n");
  actrTransactionManager->endTransaction();
}

yes = Adesk::kFalse;
if (getYOrN("\nAbort transaction two? [N] : ",
Adesk::kFalse, yes) == Acad::eOk
&& yes == Adesk::kTrue)
{
  ads_printf("\n\n$$$$$$  Aborting transaction two.\n" "$$$$$$\n");
  actrTransactionManager->abortTransaction();
  ads_printf("\nBack to separate extrusion and\n" "cylinder.");
} else {
  ads_printf("\n\n>>>>>>  Ending transaction two.\n" "<<<<<<\n");
  actrTransactionManager->endTransaction();
}
Example: Nested Transactions

```cpp
static Acad::ErrorStatus
createAndPostPoly()
{
  int nSides = 0;
  while (nSides < 3) {
    ads_initget(INP_NNEG, "");
    switch (ads_getint("Enter number of sides: ", &nSides))
    {
    case RTNORM:
      if (nSides < 3)
        ads_printf("Need at least 3 sides.");
      break;
    default:
      return Acad::eInvalidInput;
    }
  }
  ads_point center, startPt, normal;
  if (ads_getpoint(NULL, "Locate center of polygon: ", center) != RTNORM)
  {
    return Acad::eInvalidInput;
  }
  startPt[0] = center[0];
  startPt[1] = center[1];
  startPt[2] = center[2];

  // Code for aborting transaction
  if (getYOrN("Abort transaction one? [N] : ",
             Adesk::kFalse, yes) == Acad::eOk
       && yes == Adesk::kTrue)
  {
    ads_printf(" Aborting transaction one.");
    actrTransactionManager->abortTransaction();
    ads_printf("Back to just the Poly.");
  } else {
    actrTransactionManager->endTransaction();
    ads_printf(" Ending transaction one.");
  }
}
```
while (asPnt3d(startPt) == asPnt3d(center)) {
    switch (ads_getpoint(center,
               "Locate start point of polygon: ", startPt)) {
        case RTNORM:
            if (asPnt3d(center) == asPnt3d(startPt))
                ads_printf("Pick 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,
// i.e. (0, 0, 1) since we also got the center and
// start point in the current UCS. (ads_getpoint()
// returns in the current UCS.)

normal[X] = 0.0;
normal[Y] = 0.0;
normal[Z] = 1.0;

rx_uc2wc(center, center, Adesk::kFalse);
rx_uc2wc(startPt, startPt, Adesk::kFalse);
rx_uc2wc(normal, normal, Adesk::kTrue);

AcGePoint3d cen = asPnt3d(center),
    start = asPnt3d(startPt);
AcGeVector3d norm = asVec3d(normal);

AsdkPoly *pPoly = new AsdkPoly(cen, start, nSides,
                               norm, "transactPoly");
pPoly->setDatabaseDefaults(acdbCurDwg());
polyToDb(pPoly);
return Acad::eOk;
}

// Extrudes the poly to a given height.
static Acad::ErrorStatus
extrudePoly(AsdkPoly* pPoly, double height) {
    Acad::ErrorStatus es = Acad::eOk;

    // Explode to a set of lines
    AcDbVoidPtrArray lines;
poly->explode(lines);

    for (int i = 0; i < lines.size(); i++) { 
        es &= extrude Poly(p, height);
    }
    return es;
}
Example: Nested Transactions

// 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 (ads_entsel(prompt, ename, pickpt)) {
            case RTNORM:
AOK(acdbGetObjectId(objId, ename));
if (objId != checkWithThisId) {
    ads_printf("Coward! Select the proper solid.");
    continue;
}
AOK(pTransaction->getObject(pObj, objId, mode));
assert(pObj != NULL);
pSolid = AcDb3dSolid::cast(pObj);
if (pSolid == NULL) {
    ads_printf("Not a solid. Try again");
    AOK(pObj->close());
    continue;
} break;
case RTNONE:
case RTCAN:
    return Acad::eInvalidInput;
default:
    continue;
} break;
return Acad::eOk;
This chapter describes the deep clone functions, which copy an object as well as 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.

In general, to use the `AcDbDatabase::deepCloneObjects()` function in your code, you do not need to be aware of the details 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 306.

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

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.

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()`.
Ownership

Relationships between objects are stored in the object as a data member of class AcDbObjectId. 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 deep clone and wblock clone use the object ID. For more information, see “Object References” on page 165. Deep clone follows hard and soft owner connections, and wblock clone follows hard owner and pointer connections, as shown in the following figure:

<table>
<thead>
<tr>
<th>Hard owner</th>
<th>Hard pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>deep clone</td>
<td>wblock clone</td>
</tr>
<tr>
<td>wblock clone</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Soft owner</th>
<th>Soft pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>deep clone</td>
<td></td>
</tr>
</tbody>
</table>

Ownership with deep clone and wblock clone

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 308).

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.
Chapter 13  Deep Cloning

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 308.

**Typical Deep Clone Operation**

The following code excerpt shows a typical use of `AcDbDatabase::deepCloneObjects()`. The basic steps are as follows:

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`) which 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()`:

```c
void cloneSameOwnerObjects()
{
    // Step 1: Obtain the set of objects to be cloned
    ads_name sset;
    if (ads_ssget(NULL, NULL, NULL, NULL, sset) != RTNORM) {
        ads_printf("\nNothing selected\n");
        return;
    }

    // Step 2: Add obtained objectIds to list of objects
    // to be cloned
    long length;
    ads_sslength(sset, &length);
    AcDbObjectIdArray objList;
    AcDbObjectId ownerId = AcDbObjectId::kNull;
    for (int i = 0; i < length; i++) {
        ads_name ent;
        ads_ssname(sset, i, ent);
Deep Clone Basics

```cpp
AcDbObjectId objId;
acdbGetObjectId(objId, ent);

// check to be sure same owner as 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();

ads_ssfree(sset);

// Step 3: Get the objectId of the desired owner for
// the cloned objects. We'll use Model Space for
// this example.
AcDbBlockTable *pBlockTable;
acdbCurDwg()->getBlockTable(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()
//
acdbCurDwg()->deepCloneObjects(objList, modelSpaceId,
    idMap);

// Now we can go through the idMap and do whatever we'd
// like to the original and/or clone objects.
//
// For this example we'll print out the objectId's 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;
    ads_printf("ObjectId 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`:

```cpp
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 in order 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`:

```cpp
acdbCurDwg()->deepCloneObjects(mslist, modelSpaceId, idMap, Adesk::kTrue);
acdbCurDwg()->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:

```cpp
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'll need to do two ads_ssget's

    ads_printf("Select entities to be cloned to Model Space");
    ads_name ssetMS;
    ads_ssget(NULL, NULL, NULL, NULL, ssetMS);
    long lengthMS;
    ads_sslength(ssetMS, &lengthMS);
    ads_printf("Select entities to be cloned to Paper Space");
    ads_name ssetPS;
    if (ads_ssget(NULL, NULL, NULL, NULL, ssetPS) != RTNORM && lengthMS == 0)
    {
        ads_printf("Nothing selected");
        return;
    }
    long lengthPS;
    ads_sslength(ssetPS, &lengthPS);
```
// Step 2: Add obtained objectIds to 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;
    ads_ssnname(ssetMS, i, ent);
    AcDbObjectId objId;
    acdbGetObjectId(objId, ent);
    // check to be sure same owner as 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();
}
ads_ssfree(ssetMS);

// For Paper Space
ownerId = AcDbObjectId::kNull;
AcDbObjectIdArray objListPS;
for (i = 0; i < lengthPS; i++) {
    ads_name ent;
    ads_ssnname(ssetPS, i, ent);
    AcDbObjectId objId;
    acdbGetObjectId(objId, ent);
    // check to be sure same owner as 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();
}
ads_ssfree(ssetPS);
// Step 3: Get the objectId of the desired owners for the cloned objects. We're using Model Space and Paper Space for this example.

AcDbBlockTable *pBlockTable;
acdbCurDwg()->getBlockTable(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()
acdbCurDwg()->deepCloneObjects(objListMS, modelSpaceId, idMap, Adesk::kTrue);
acdbCurDwg()->deepCloneObjects(objListPS, paperSpaceId, idMap);

// Now we can go through the idMap and do whatever we'd like to the original and/or clone objects.
// For this example we'll print out the objectId's 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;
    ads_printf("ObjectId 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 that you can override the deepClone() and wblockClone() methods for your own custom objects and entities.
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 a 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 `deepClone()` 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).

Case 1

As an example of Case 1, suppose you have Entity A1, which contains a pointer reference to Entity B1. Both Entity A1 and Entity B1 are selected to be cloned. Before translation, EntityA2 still refers to Entity B1. After translation, EntityA2 is updated to refer to EntityB2.
**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.

![Case 2: After Cloning](image)

Case 2: After Translating

![Case 2: After Translating](image)

**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.

![Case 3: After Cloning](image)

Case 3: After Translating

![Case 3: After Translating](image)
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 325.

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.

```cpp
// This example demonstrates a way to handle objects in
// the NamedObjects dictionary for WBLOCK and INSERT.
// Our object is an AcDbDictionary which is called
// "AsdkDictionary" in the NamedObjects 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;
```
Implementing deepClone() for Custom Classes

// 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
      // NamedObject 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 which
      // 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);
Chapter 13  Deep Cloning

```cpp
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.  So, first we must find out if
    // the destination NamedObjects dictionary has
    // our dictionary.
    //
    AcDbDictionary *pDestNamedDict;
    pTo->getNamedObjectsDictionary(pDestNamedDict,
        AcDb::kForWrite);
    // Since INSERT does not clone the destination
    // NamedObjects dictionary, we can use getAt() on it.
    //
    es = pDestNamedDict->getAt(kpDictionary, dictId);
    // If our dictionary does not yet exist in the
    // NamedObjects 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();
```
Implementing deepClone() for Custom Classes

// 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 objectIds are not in flux, and we can use getAt() and setAt() on it.

AcDbDictionary *pDestDict;
acdbOpenObject(pDestDict, dictId, AcDb::kForWrite);

AcDbObject *pObj, *pObjClon;
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 idPairs into the idMap 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 "*". It will become something like, "**S04".
    if (pDestDict->getAt(pName, dictId) == Acad::eKeyNotFound)
        pDestDict->setAt(pName, pObjClone, dictId);
    else
        pDestDict->setAt("*", pObjClone, dictId);
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.

The basic steps in the cloning stage are as follows:

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 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.) Then 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:

```cpp
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.
    //
    AcDbIdPair idPair(objectId(), (AcDbObjectId)NULL,
                      Adesk::kFalse, isPrimary);
    if (idMap.compute(idPair) && (idPair.value() != NULL))
        return Acad::eOk;
```
Chapter 13  Deep Cloning

// STEP 1:
// Create the clone
//
AsdkPoly *pClone = (AsdkEntity*)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,
// we know that we are derived from AcDbEntity, so we
// can expect our owner to be an AcDbBlockTableRecord.
// unless we have set up an ownership relationship with
// another of our objects. In that case, we need to
// establish how we connect to that owner in our own
// way. 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 our object
    // other than with an AcDbBlockTableRecord.
    //
    pOwner->database()->addAcDbObject(pClone);
    pClone->setOwnerId(pOwner->objectId());
}

// STEP 3:
// Now we copy our contents to the clone. This is done
// using an AcDbDeepCloneFiler. This filer keeps a
// list of all AcDbHardOwnershipIds and
// AcDbSoftOwnershipIds we, and any classes we derive
// from, have. 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->setAcDbObjectIdIsInFlux();

// STEP 6:
// Add the new information to the idMap. We can use
// the idPair 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 we now
    // set "isPrimary" 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 "cheapCloned". In this case, they are
    // "moved" instead of cloned. The result is that
    // pSubObject and pClonedSubObject will point to
    // the same object. So, we only want to close
    // pSubObject if it really is a different object
    // than its clone.
    //
    if (pSubObject != pClonedSubObject)
        pSubObject->close();
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 314.

The wblockClone() function uses a flier 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 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.
An example of a case where you would need to pass in the database as the owner of an object is 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 `Layer1`, `Layer1` 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:

<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
<th>isCloned</th>
<th>isPrimary</th>
<th>isOwnerXlated</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTR1</td>
<td>BTR2</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>SPH1</td>
<td>SPH2</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>LT1</td>
<td>LT2</td>
<td>TRUE</td>
<td>FALSE</td>
<td>*</td>
</tr>
<tr>
<td>LTR1</td>
<td>LTR2</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE†</td>
</tr>
</tbody>
</table>
Implementing `deepClone()` for Custom Classes

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 LayerTable1 to LayerTable2.

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 325.

```cpp
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, we set it here as well.
    //
    pClonedObject = NULL;

    // If this object is in the idMap and is already
    // cloned, then return.
    //
    AcDbIdPair idPair(objectId(), (AcDbObjectId)NULL,
                      Adesk::kFalse, isPrimary);
    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. So, 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 idMap.

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 which must be cloned during a Wblock operation.

AcDbWblockCloneFiler filer;
dwgOut(&filer);

STEP 4:
Rewind the file 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->setAcDbObjectIdIsInFlux();

// STEP 6:
// Add the new information to the idMap. We can use
// the idPair 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's 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;
  }

  // We can find out if this is an AcDbHardPointerId
  // verses an AcDbHardOwnershipId, by seeing 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 we now set "isPrimary" 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->objectId() == 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 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;

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 it opens the model space block table record of the source database, not the destination database. The untranslated reference in the destination database’s block table is still referring to the model space of the source database.

SampleReactor::otherWblock(AcDbDatabase* pDestDb,
                           AcDbIdMapping& idMap,
                           AcDbDatabase* pSrcDb)
{
    AcDbBlockTable* pDestBlockTable;
    AcDbBlockTableRecord* pDestBTR;
    pDestDb->getBlockTable(pDestBlockTable, AcDb::kForRead);
    pDestBlockTable->getAt(AcDb::kForWrite,
                          pDestBTR);
    // pDestBTR is pointing to pSrcDb database’s Model Space !
    // This code is incorrect!
To find the destination model space, you must look it up in the ID map:

```c++
void AsdkWblockReactor::otherWblock(AcDbDatabase* pDestDb,
              AcDbIdMapping& idMap,
              AcDbDatabase* pSrcDb)
{
    AcDbBlockTable *pDestBlockTable;
    AcDbBlockTableRecord *pDestBTR;

    pDestDb->getBlockTable(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!
}
```

**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()`, this faster form of cloning (called “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 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.
Implementing deepClone() for Custom Classes

5 endWblock(). The entities have now been transformed, and the model and paper space origins have been set. The new database is complete but has not yet been saved.

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
Chapter 13 Deep Cloning

kDcXrefBind XREF Bind and XBIND
kDcSymTableMerge 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.

```cpp
// 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 which are referenced by entities being Wblocked will be included (Wblock's default behavior).

// AsdkEdReactor is derived from AcEditorReactor

void AsdkEdReactor::beginDeepCloneXlation(AcDbIdMapping& idMap, Acad::ErrorStatus*)
{
    // getYorN() is a user defined function which prompts for a user response and returns a boolean.
    if (idMap.deepCloneContext() == AcDb::kDcWblock && getYorN("Wblock all Text Styles"))
    {
        AcDbDatabase *pOrigDb, *pDestDb;
        idMap.origDb(pOrigDb);
        idMap.destDb(pDestDb);

        AcDbTextStyleTable *pTsTable;
        pOrigDb->getTextStyleTable(pTsTable, AcDb::kForRead);
        AcDbTextStyleTableIterator *pTsIter;
        pTsTable->newIterator(pTsIter);
        AcDbTextStyleTableRecord *pTsRecord;
        AcDbObject *pClonedObj;
        for (; !pTsIter->done(); pTsIter->step())
        {
            pTsIter->getRecord(pTsRecord, AcDb::kForRead);
        }
    }
```
Implementing deepClone() for Custom Classes

```c++
// We don't need 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;
pTsRecord->wblockClone(pDestDb, pClonedObj, idMap);
pTsRecord->close();
if (pClonedObj != NULL)
    pClonedObj->close();
}
delete pTsIter;
pTsTable->close();
```
C++ class definitions are fixed at compile time. However, if you write a file translator or an editing command that needs to operate on a number of existing AutoCAD classes, you would need to redefine all the existing classes to include the new translator or editing functions. You would then have to recompile the C++ library as well as all applications that use it.

With the ARX protocol extension that is described in this chapter, you can add functions to existing AutoCAD classes at run time without recompiling. The protocol extension mechanism uses the ARX class descriptor object, described in chapter 8, “Deriving a custom ARX Class” Whenever a class is registered with ARX, a class descriptor object is created. This object describes the class and is also used as a space-holder for another set of objects that extend the protocols of the class. An ARX class can have any number of protocol extension objects associated with it.
Defining Protocol Extensions

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 \texttt{AcDbEntity}, and specific implementations are defined for \texttt{AcDbCircle} and \texttt{AcDbRegion}.

The example serves as a model for more complex (and realistic) uses of the ARX protocol extension mechanism. The base class for this protocol extension, called \texttt{AsdkEntTemperature}, is derived from \texttt{AcRxObject}. This class defines the virtual functions that will be inherited by the derived protocol extension classes, \texttt{AsdkDefaultTemperature}, \texttt{AsdkCircleTemperature}, and \texttt{AsdkRegionTemperature}. In this example, there is only one function: \texttt{reflectedEnergy()}. The class hierarchy for the protocol extension classes is shown in the following figure:

![Class Hierarchy Diagram]

The first step in using protocol extension is to declare and define each of the protocol extension classes. The base class, \texttt{AsdkEntTemperature}, is an abstract base class that is defined using the \texttt{ACRX\_NO\_CONS\_DEFINE\_MEMBERS()} macro. This class will eventually be registered as part of the ARX class hierarchy (see the following section, “Registering Protocol Extensions”).

The child classes are defined using standard C++ syntax for deriving new classes. These classes should not be registered in the ARX class hierarchy, so you don’t need to use the ARX macros for them.

For each class, you implement the functions that are the protocol extension. In this example, each class has only one function, \texttt{reflectedEnergy()}, which calculates a temperature for the entity.
Registering Protocol Extensions

There are two main parts to registering the protocol extensions with your application:

1. Initialize your new protocol extension parent class and add it to the run-time class hierarchy as shown in the following example:

   ```cpp
   AsdkEntTemperature::rxInit();
   acrxBuildClassHierarchy();
   ```

   These function calls are required for any new ARX class, as described in chapter 8, “Deriving a Custom ARX 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:

   ```cpp
   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 run time, ARX constructs a class descriptor object structure that includes the basic ARX class hierarchy as well as the protocol extension objects associated with the ARX 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, a recursive search is made to find a protocol extension object. It is also recommended that you associate the default protocol extension class with `AcRxObject` or with some other class at the top of the ARX class hierarchy such as `AcDbEntity` (in this example) or `AcDbObject`. If AutoCAD walks up the entire tree and does not find a protocol extension object, it aborts.

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 ARX class dictionary and delete the class descriptor object for your protocol extension class (see the `unloadApp()` function in the chapter example).
Using Protocol Extensions in an Application

To use protocol extension, 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:

```cpp
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:

```cpp
double eTemp = ACRX_X_CALL(pEnt, AsdkEntTemperature)->reflectedEnergy(pEnt);
```

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 ARX module interface functions: `initApp()`, `unloadApp()` and `acrxEntryPoint()`.

```cpp
class AsdkEntTemperature : public AcRxObject
{
// AsdkEntTemperature Protocol extension abstract base
// class. Notice that this is the lowest level that uses
// the ACRX macros.

public:
    ACRX_DECLARE_MEMBERS(AsdkEntTemperature);
    virtual double reflectedEnergy(AcDbEntity*) const = 0;
};
ACRX_NO_CONS_DEFINE_MEMBERS(AsdkEntTemperature, AcRxObject);
```
class AsdkDefaultTemperature : public AsdkEntTemperature
//
// 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.
//
{
public:

  virtual double reflectedEnergy(AcDbEntity* pEnt) const;
};

double
AsdkDefaultTemperature::reflectedEnergy(  
  AcDbEntity* pEnt) const
{
  ads_printf("This entity has no area, and no reflection.\n");
  return -1.0;
}

class AsdkRegionTemperature : public AsdkEntTemperature
//
// AsdkEntTemperature implementation for Regions.
//
{
public:

  virtual double reflectedEnergy(AcDbEntity* pEnt) const;
};

double
AsdkRegionTemperature::reflectedEnergy(  
  AcDbEntity* pEnt) const
{
  AcDbRegion *pRegion = AcDbRegion::cast(pEnt);

  if (pRegion == NULL)
    ads_printf("The impossible has happened!\n");

  // Compute reflected energy as region area multiplied
  // by a dummy constant.
  //
  double retVal;
  if (pRegion->getArea(retVal) != Acad::eOk)
    return -1.0;

  return retVal * 42.0;
}
class AsdkCircleTemperature : public AsdkEntTemperature  
// AsdkEntTemperature implementation for circles. 
{  
public:  

    virtual double reflectedEnergy(AcDbEntity* pEnt) const;  
};  

    double  
AsdkCircleTemperature::reflectedEnergy(  
      AcDbEntity* pEnt) const  
{  
    AcDbCircle *pCircle = AcDbCircle::cast(pEnt);  
    // Compute reflected energy in 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  
enenergy()  
{  
    AcDbEntity *pEnt;  
    AcDbObjectId pEntId;  
    ads_name en;  
    ads_point pt;  
    if (ads_entsel("Select an Entity: ", en, pt)  
        != RTNORM)  
    {  
        ads_printf("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);  
    ads_printf("Energy == %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 as well as adding protocol extension
// objects to classes in and custom classes to the ACRX
// runtime class hierarchy.

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 that 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* pkt)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(pkt);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
    }
    return AcRx::kRet
This chapter describes how to use the Microsoft Foundation Class (MFC) library as part of an ARX application. It assumes you are familiar with ARX applications and how to build them. For information on MFC, see the appropriate Microsoft documentation.
Chapter 15 Using MFC With an ARX Application

General Guidelines

Because an ARX application is a dynamic linked library (DLL), you need to observe the following Microsoft guidelines:

- Define the symbols \_WINDLL and \_USRDLL so that your application will compile successfully.
  
  /D\_WINDLL signifies that the compilation is for a DLL.
  
  /D\_USRDLL specifies that you are building a DLL using the statically linked version of MFC.

- Link your ARX application with the static MFC library (nafxdwd.lib or nafxdw.lib) along with the version of the C run-time library called libcmt.lib.

The following MFC functions are not available when you link with the static version of the library:

- \_\_WINDLL

- \_\_USRDLL

Creating a Project

MFC provides an easy way for you to add dialog boxes to your ARX application. Although some restrictions result from ARX applications being DLLs, you can use the Visual C++ AppWizard to create your application and the ClassWizard to maintain it.

The Visual C++ AppWizard is the easiest way to create an MFC project. In the File menu, select New, then choose Project from the New dialog. From the New Project dialog, select either “MFC AppWizard (dll)” or “Dynamic-Link Library.” Choose Win32 under Platforms. Note that if you select Dynamic-Link Library instead of MFC AppWizard (dll), you lose the use of some wizards that make MFC programming more productive.

When you choose Create, Visual C++ displays the AppWizard - Step 1 of 1 dialog. In most cases, you can choose the options you prefer, but it is extremely important to enable the Use MFC in a static library option. If you use MFC in a DLL, your application will crash and perhaps bring AutoCAD down along with it.

Note: If the AppWizard dialog starts with the Single Document/Multiple Documents/Dialog-Based selection, then you have mistakenly chosen to create an EXE instead of a DLL.
Initializing MFC

To initialize MFC, you need to call the MFC internal function `RawDllMain()` as soon as your application is loaded into AutoCAD and again when your application is unloaded. The `RawDllMain()` function creates and initializes the ARX core state object, which must exist before you create the application object (“the App”).

The recommended approach is to define a C++ class that includes a call to `RawDllMain()` with `DLL_PROCESS_ATTACH` as one of its arguments when the ARX application is loaded, and a call to `RawDllMain()` with `DLL_PROCESS_DETACH` as one of its arguments when the ARX application is unloaded. Create a global object of this class before the application object is created:

```cpp
class CLoadMFC
{
public:
    CLoadMFC()
    { RawDllMain(_hdllInstance, DLL_PROCESS_ATTACH, NULL); }
    ~CLoadMFC()
    { RawDllMain(_hdllInstance, DLL_PROCESS_DETACH, NULL); }
};
CLoadMFC CLoadMFC;
```

In the sample program at the end of this chapter, this code is included in the section “Application File (MFCTEMPL.CPP).”

**Note** These instructions are for Microsoft Visual C++ Version 2.2 and may vary with future releases of the Visual C++ product. For version 4.0, you do not need to call `RawDllMain()`. Check the current ARX readme file for instructions on using ARX with subsequent releases of Visual C++.

Other DLL Initialization

To perform other DLL initialization, you then need to call `DllMain()` in the `acrxEEntryPoint()` function, as shown in the following example. `DllMain()` is defined in the static MFC library.

```cpp
extern "C" AcRx::AppRetCode acrxEEntryPoint( AcRx::AppMsgCode msg, void* );
{
    switch( msg )
    {
    case AcRx::kInitAppMsg:
        DllMain(_hdllInstance, DLL_PROCESS_ATTACH, NULL);
        // plus application-specific initialization
        break;
    case AcRx::UnloadAppMsg:
        DllMain(_hdllInstance, DLL_PROCESS_DETACH, NULL);
        break;
    ```
Chapter 15  Using MFC With an ARX Application

```cpp
default:
    break;
}
return AcRx::kRetOK;
}

In the sample code at the end of this chapter, the functions InitMFC() and EndMFC() (defined in mfctempl.cpp) contain the calls to DllMain().

The Application Object

The application object is derived from CWinApp. It overrides the parent class member functions InitInstance() and ExitInstance(). In the example presented in this chapter, these two functions simply call their parents’ member functions. (See mfctempl.cpp.)

Memory Allocation

No changes are required if you are using the released version of the static MFC library inside an ARX application. However, with the debugging version of the MFC library, the Memory Debug tool may cause conflicts with the AutoCAD memory allocator. To solve the problem, override the following allocators in your ARX application:

```cpp
#ifdef _DEBUG
    void *operator new (size_t nSize, LPCSTR lpszFileName, int nLine)
    {
        return operator new(nSize);
    }
    void *CObject::operator new(size_t nSize)
    {
        return ::operator new(nSize);
    }
    void *CObject::operator new (size_t nSize, LPCSTR lpszFileName, int nLine)
    {
        return ::operator new(nSize);
    }
    void CObject::operator delete(void *pbData)
    {
        delete pbData;
    }
#endif // _DEBUG
```
When you overload \texttt{operator new} in this manner, it will be a multiply-defined symbol. Under normal conditions, the linker treats this as an error. To turn these errors into warnings, go to the Link section of the Project Settings dialog. Choose Customize for the Category of the setting and enable the Force File Output option. When you build your application, you will get several linker warnings about multiply-defined symbols, which you can ignore.

\textbf{Modeless Dialogs}

Because 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 \texttt{WM\_ACAD\_KEEPFOCUS} window message, which is defined in \texttt{adscodes.h} as 1001. When your dialog gets this message, it must return \texttt{TRUE} if it should keep focus. If the response to this message is \texttt{FALSE} (which is also the default), then your dialog will lose focus as soon as the user moves the mouse pointer off of the dialog's window.

You can do this with the dialog's message map, and an \texttt{ON\_MESSAGE()} declaration as shown:

\begin{verbatim}
BEGIN_MESSAGE_MAP( HelloDlg, CDialog )
   ON_COMMAND( IDCLOSE, OnClose )
   ON_COMMAND( IDC_DRAW_CIRCLE, OnDrawCircle )
   ON_MESSAGE( WM_ACAD_KEEPFOCUS, onAcadKeepFocus )
END_MESSAGE_MAP()
\end{verbatim}

In this example, the application's dialog class is \texttt{HelloDlg}, which is derived from \texttt{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 \texttt{keepTheFocus()}, which returns \texttt{TRUE} if your dialog wants to keep the input focus and \texttt{FALSE} if the dialog is willing to yield the focus to AutoCAD. Your example message handler would then be as follows.

\begin{verbatim}
afx_msg LONG HelloDlg::onAcadKeepFocus( UINT, LONG )
{
   return keepTheFocus() ? TRUE : FALSE;
}
\end{verbatim}
Using AutoCAD’s Frame and View Window Handle

You may need to obtain the handle to the AutoCAD frame window—for example, when you want a dialog box or other window to use AutoCAD’s window as the parent. Use the following function to obtain the handle to the AutoCAD frame window:

```cpp
extern HWND adsw_acadMainWnd();
```

This prototype is not in any of the header files, so you need to prototype it explicitly.

To obtain a pointer to the AutoCAD CWnd object, you can call `CWnd::FromHandle()` using the handle returned by `adsw_hwndAcad()` as its argument. This pointer to the CWnd object is required when you want to create a dialog box with AutoCAD as its parent.

To obtain the handle of the AutoCAD document view window, use the following function:

```cpp
extern HWND adsw_acadDocWnd();
```

Example

This sample application creates a modal dialog box with OK and Cancel buttons. The parent window is AutoCAD.

This chapter shows three files: the source and header files that perform MFC initialization, and the ARX application file. See the online Samples directory for the complete set of files related to this example.

The application object is derived from `CWinApp`, as shown in `mfctempl.h`. This class (CMFCTemplateApp) overrides the `InitInstance()` and `ExitInstance()` functions of `CWinApp`, as shown in `mfctempl.h` and `mfctempl.cpp`.

Main Header File (`MFCTEMPL.H`)

This file declares the MFC initialization routines.

```cpp
#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"

// CMFCTemplateApp
// See MFCTemplate.cpp for the implementation of this class

class CMFCTemplateApp : public CWinApp {
```
public:
CMFCTemplateApp();
virtual ~CMFCTemplateApp();
// Overrides
// ClassWizard generated virtual function overrides
//}}AFX_VIRTUAL(CMFCTemplateApp)
public:
virtual BOOL InitInstance();
virtual int ExitInstance();
//}}AFX_VIRTUAL
//{{AFX_MSG(CMFCTemplateApp)
// NOTE: the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

// CDiaTest dialog
class CDiaTest : public CDialog
{
// Construction
public:
CDiaTest(CWnd* pParent = NULL); // standard constructor

// Dialog Data
//}}AFX_DATA(CDiaTest)
enum { IDD = IDD_TEST }; // NOTE: the ClassWizard will add data members here
//}}AFX_DATA

// Overrides
//}}AFX_VIRTUAL(CDiaTest)
protected:
virtual void DoDataExchange(CDataExchange* pDX);
//}}AFX_VIRTUAL

// Implementation
protected:

// Generated message map functions
//}}AFX_MSG(CDiaTest)
// NOTE: the ClassWizard will add member functions here
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
Chapter 15 Using MFC With an ARX Application

**Application File (MFCTEMP1.CPP)**

This file defines the initialization routines for MFC.

```cpp
#include "stdafx.h"
#include "MFCTempl.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

extern HWND adsw_acadMainWnd();
extern "C" HINSTANCE _hdllInstance;
static HWND hWndACAD = NULL;
extern "C" BOOL WINAPI RawDllMain(HINSTANCE, DWORD dwReason, LPVOID);
extern "C" BOOL APIENTRY DllMain(HINSTANCE, DWORD, LPVOID);

class CLoadMFC
{
public:
  CLoadMFC() { RawDllMain(_hdllInstance, DLL_PROCESS_ATTACH, NULL); }
  ~CLoadMFC() { RawDllMain(_hdllInstance, DLL_PROCESS_DETACH, NULL); }
};

CLoadMFC CLoadMFC;

void InitMFC()
{
  DllMain(_hdllInstance, DLL_PROCESS_ATTACH, NULL);
  hWndACAD = /* GetActiveWindow() */ adsw_acadMainWnd();
}

void EndMFC()
{
  DllMain(_hdllInstance, DLL_PROCESS_DETACH, NULL);
}

void MFCTest()
{
  CDiaTest Diatest(CWnd::FromHandle(hWndACAD));
  Diatest.DoModal();
}

#endif

// CMFCTemplateApp
BEGIN_MESSAGE_MAP(CMFCTemplateApp, CWinApp)
 //{{AFX_MSG_MAP(CMFCTemplateApp)
  // NOTE - the ClassWizard will add and remove mapping macros here.
  //    DO NOT EDIT what you see in these blocks of generated code!
 //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// CMFCTemplateApp construction
CMFCTemplateApp::CMFCTemplateApp()
{
}

CMFCTemplateApp::~CMFCTemplateApp()
{
}

// The one and only CMFCTemplateApp object
```
CMFCTemplateApp theApp;

BOOL CMFCTemplateApp::InitInstance()
{
    TRACE("InitInstance MFCTemplate\n");
    return CWinApp::InitInstance();
}

int CMFCTemplateApp::ExitInstance()
{
    TRACE("ExitInstance MFCTemplate\n");
    return CWinApp::ExitInstance();
}

#ifdef _DEBUG
void *operator new (size_t nSize, LPCSTR lpszFileName, int nLine)
{
    return operator new(nSize);
}

void* CObject::operator new(size_t nSize)
{
    return ::operator new(nSize);
}

void *CObject::operator new (size_t nSize, LPCSTR lpszFileName, int nLine)
{
    return ::operator new(nSize);
}

void CObject::operator delete(void *pbData)
{
    delete pbData;
}
#endif // _DEBUG

// CDiaTest dialog
CDiaTest::CDiaTest(CWnd* pParent /*=NULL*/)
    : CDialog(CDiaTest::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDiaTest)
    // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void CDiaTest::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDiaTest)
    // NOTE: The ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CDiaTest, CDialog)
    //{{AFX_MSG_MAP(CDiaTest)
    // NOTE: The ClassWizard will add message map macros here
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// CDiaTest message handlers
Main ARX File (RXTEMPLT.CPP)

This is the main ARX application file, which contains the acrxE EntryPoint() function.

```cpp
#include "stdafx.h"
#include <aced.h>
#include <adslib.h>

// Standard C Test function
void rxTest()
{
    ads_printf( "\nTest function called\n" );
    MFCTest();
}

// Rx interface
void initApp()
{
    InitMFC();
    ads_printf( "\nLoading RXTEMPLATE Sample ...\n" );
    acedRegCmds->addCommand( "RXTEMPLATE",      // Group name
                             "ESC_RXTEST",      // Global function name
                             "RXTEST",          // Local function name
                             ACRX_CMD_MODAL,    // Type
                             &rxTest );         // Function pointer
    ads_printf( "\nOK!\n" );
}

void unloadApp()
{
    // Remove the command group because we are unloading
    // acedRegCmds->removeGroup( "RXTEMPLATE" );
    EndMFC();
}

extern "C" AcRx::AppRetCode acrxEEntryPoint( AcRx::AppMsgCode msg, void* )
{
    switch( msg )
    {
    case AcRx::kInitAppMsg:
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
        break;

    default:
        break;
    }
    return AcRx::kRetOK;
}
```
APPENDIXES

APPENDIX A ■ MIGRATING ADS PROGRAMS TO ARX

APPENDIX B ■ USING THE GEOMETRY LIBRARY
MIGRATING ADS PROGRAMS TO ARX

To simplify the migration of ADS applications to the ARX program environment, the ADS library was ported to the ARX program environment, and the ARX version is almost identical to the ADS version. This appendix provides a comparison of how programs are loaded in both ADS and ARX and includes a sample program that has been ported from ADS to ARX.
Reasons for Migrating to ARX

Applications that frequently communicate with AutoCAD through the ADS library or other calls, as opposed to being bound by heavy computations, run faster in the ARX program environment than in the ADS program environment. To take advantage of the powerful API libraries developed for the ARX program environment, existing ADS applications can be ported to ARX.

In some cases, you may want to migrate only the portion of an ADS application that interacts heavily with AutoCAD, leaving less performance-intensive code in AutoLISP or ADS. The ARX application can be invoked like an ADS application with the AutoLISP function call or by `ads_invoke()`, but it also can operate independent of AutoLISP or ADS.

The same set of ADS library functions is accessed by applications from both the ARX and the ADS program environments, but certain functions are reserved for use by either ADS or ARX applications.

acrxEntryPoint

An ARX 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:

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

The first parameter, a data member of class `AcRx` called `msg`, represents the message sent from the ARX 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 RSRLT and RSERR.

AutoCAD calls into the ARX module `acrxEntryPoint()` to pass messages to the application. All requests to invoke functions through `ads_defun()` are made by `acrxEntryPoint()`, simplifying the migration of ADS programs to the ARX program environment.

ARX 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 ARX application source file.

`adesk.h` Contains standard definitions for the ARX program environment.

`rxdefs.h` Establishes an ARX application and interacts with AutoCAD through `acrxEEntryPoint()`.

`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 ARX application source files.

`adsdlg.h` Contains proteus-related declarations for creating dialog boxes.

`ol_errno.h` Contains symbolic codes for the error values used by the AutoCAD system variable ERRNO. These codes are shown in appendix A, “Error Codes,” of the AutoCAD ADS Programmer’s Guide.

`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 ARX application doesn’t 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 ARX

Application loading in the ARX 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 ARX 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 ARX 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.

ARX and ADS application loading are different in the following ways.

- When a drawing is closed or when another drawing is opened, ARX applications are not unloaded. An ARX application is unloaded when AutoCAD is terminated or when the application has no dependents and ads_arxunload() or an equivalent function is called.

- In the ARX 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 ARX application is initialized or unloaded.

- In the ADS program environment, a drawing is present when a kInitAppMsg or kUnloadAppMsg is received.

  **Note** In the ARX program environment, do not assume that a drawing is present when a kInitAppMsg or kUnloadAppMsg is received.

- In the ARX program environment, call (arxload) or ads_arxload() to load applications. In the ADS program environment, call (xload) or ads_xload() to load applications.

- ARX applications listed in acad.rx are loaded automatically when AutoCAD is invoked, and unlike ADS applications, the ARX applications are initialized before a drawing is present and before the ADS or AutoLISP 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 Applications in the ARX Program Environment

ADS applications are implemented as DLL and have a default file extension of .arx. For instructions on building ADS applications in the ARX program environment, see the readdev.hlp file.

The procedure described in the readdev.hlp file for debugging ADS applications in the ADS program environment also applies to the ARX program environment.

Sample Application

Compare the sample applications of the factorial programs distributed with AutoCAD. The fact.cc program is for the ARX program environment, while fact.c is for the ADS program environment. The following version of the fact.cpp declares and implements acrxE EntryPoint(); otherwise, fact.cpp is almost identical to fact.c. The function acrxEEntryPoint() replaces the main in fact.c.

```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 *)); };
Appendix A  Migrating ADS Programs to ARX

/* 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*/
    {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 ARX 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();
        break;
    }
    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 (!ads_defun(func_table[i].func_name, (short)i))
            return RTERROR;
    }
    return RTNORM;
}
```c
/*-----------------------------------------------------------------------*/
/* 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 = ads_getfuncode()) < 0 || val >= ELEMENTS(func_table)) {
        ads_fail(/*MSG2*/"Received nonexistent function code.");
        return RTERROR;
    }

    /* Fetch the arguments, if any. */
    rb = ads_getargs();

    /* Call the handler and return its success-failure status. */
    val = (*func_table[val].func)(rb);
    ads_relrb(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 {
        ads_fail(/*MSG3*/"Argument should be an integer.");
        return RTERROR;
    }

    if (x < 0) {                     /* Check argument range */
        ads_fail(/*MSG4*/"Argument should be positive.");
        return RTERROR;
    } else if (x > 170) {            /* Avoid floating-point overflow */
        ads_fail(/*MSG5*/"Argument should be 170 or less.");
        return RTERROR;
    }

    ads_retreal(rfact(x));          /* Call the function itself, and
         return the value to AutoLISP */
    return RTNORM;
}
```
Appendix A  Migrating ADS Programs to ARX

#include "ads.h"

/* 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 {
        ads_fail(/*MSG6*/"Argument should be a real or integer value.");
        return RTERROR;
    }
    if (x < 0) {                      /* Check argument range */
        ads_fail(/*MSG7*/"Argument should be positive.");
        return RTERROR;
    }
    ads_retreal(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;
    c1 = 0.0;
while ((c != cl) && (n-- > 0)) {
    y = c;
    cl = c;
    c = (y - x / y) / 2;
}
return y;

**ARX-Exclusive Data Type**

**ACAD_GRAPHICS** is a new data type defined at run time for extended entity data in the ARX 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.”
This appendix 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 ARX. The examples included in this appendix show how to use the following:

- Point, vector, and matrix classes
- Linear algebra operations in the line and plane classes
- Complex geometry such as NURBS curves and surface
Overview

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; although 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 in the following figure.

```
AcGePoint2d
AcGeVector2d
AcGeMatrix2d
AcGeScale2d
AcGePoint2dArray
AcGeVector2dArray

AcGePoint3d
AcGeVector3d
AcGeMatrix3d
AcGeScale3d
AcGePoint3dArray
AcGeVector3dArray

AcGeTol
AcGeInterval
AcGeCurveBoundary
AcGeDoubleArray
AcGeKnotVector

AcGeEntity2d
AcGePointEnt2d
    AcGePosition2d
    AcGePointOnCurve2d

AcGeCurve2d
    AcGeLinearEnt2d
        AcGeLine2d
        AcGeRay2d
        AcGeLineSeg2d
        AcGeCircArc2d
        AcGeEllipArc2d
    AcGeSplineEnt2d
        AcGeCubicSpline2d
        AcGeNurbCurve2d
        AcGePolyLine2d
        AcGeExternalCurve2d
```

AcGe library class hierarchy (part 1 of 2)
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`.

### AcGe library class hierarchy (part 2 of 2)

<table>
<thead>
<tr>
<th>Class</th>
<th>(3D entities)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AcGeEntity3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGePointEnt3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGePosition3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGePointOnCurve3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGePointOnSurface</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeCurve3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeLinearEnt3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeLine3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeRay3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeLineSeg3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeArc3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeElliptArc3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeExternalArc3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeSplineEnt3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeCubicSpline3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeNurbCurve3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGePolyLine3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeAugPolyLine3d</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeSurface</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGePlanarEnt</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGePlane</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeBoundedPlane</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeCylinder</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeCone</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeSphere</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeTorus</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeNurbSurface</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeExternalSurface</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeOffsetSurface</code></td>
<td></td>
</tr>
<tr>
<td><code>AcGeExternalBoundedSurface</code></td>
<td></td>
</tr>
</tbody>
</table>
Run-time 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 they 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 functions defined by some of the header files.

**Global functions and header files**

<table>
<thead>
<tr>
<th>Header file</th>
<th>Global functions defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>gepnt2d.h</td>
<td><code>AcGePoint2d::kOrigin</code></td>
</tr>
<tr>
<td>gemat2d.h</td>
<td><code>AcGeMatrix2d::kIdentity</code></td>
</tr>
<tr>
<td>gevec2d.h</td>
<td><code>AcGeVector2d::kIdentity</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGeVector2d::kXAxis</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGeVector2d::kYAxis</code></td>
</tr>
<tr>
<td>geline2d.h</td>
<td><code>AcGeLine2d::kXAxis</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGeLine2d::kYAxis</code></td>
</tr>
<tr>
<td>gepnt3d.h</td>
<td><code>AcGePoint3d::kOrigin</code></td>
</tr>
<tr>
<td>gemat3d.h</td>
<td><code>AcGeMatrix3d::kIdentity</code></td>
</tr>
<tr>
<td>gevec3d.h</td>
<td><code>AcGeVector3d::kIdentity</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGeVector3d::kXAxis</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGeVector3d::kYAxis</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGeVector3d::kZAxis</code></td>
</tr>
<tr>
<td>geline3d.h</td>
<td><code>AcGeLine3d::kXAxis</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGeLine3d::kYAxis</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGeLine3d::kZAxis</code></td>
</tr>
<tr>
<td>geplane.h</td>
<td><code>AcGePlane::kXYPlane</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGePlane::kYZPlane</code></td>
</tr>
<tr>
<td></td>
<td><code>AcGePlane::kXZPlane</code></td>
</tr>
<tr>
<td>gegbl.h</td>
<td><code>AcGeContext (gOrthoVector)</code></td>
</tr>
</tbody>
</table>
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.

```cpp
void setEqualPoint ( double );
void setEqualVector ( double );
```

The AcGeTol class also provides two functions for obtaining the tolerance for points and vectors:

```cpp
double equalPoint() const;
double equalVector() const;
```

The `equalPoint` and `equalVector` tolerance values are used as follows:

- Two points p1 and p2 are equal if
  \[(p1 - p2).length() \leq \text{equalPoint}\]
- Two vectors v1 and v2 are equal if
  \[(v1 - v2).length() \leq \text{equalVector}\]
- Two vectors v1 and v2 are parallel if
  \[(v1/v1.length() - v2/v2.length()).length() < \text{equalVector} OR (v1/v1.length() + v2/v2.length()).length() < \text{equalVector}\]
- Two vectors v1 and v2 are perpendicular if
  \[\text{abs}((v1.dotProduct(v2))/(v1.length() * v2.length()))) < \text{equalVector}\]
- Two lines or rays are parallel (perpendicular) if their directional vectors are parallel (perpendicular).
- Two lines are equal if they have points within \text{equalPoint} and they are parallel.

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:

```c++
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:

```c++
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:

```c++
p1 += p2.asVector();
```

The following are examples of how to obtain the negative of a vector:

```c++
v2 = -v1;           // set v2 to negative of v1
v1.negate();        // equivalent to v1 = -v1;
```

The following are examples of different ways to scale a vector:

```c++
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:

```c++
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):

```c++
angle = v1.angleTo(v2,v3);
```
The following functions return a Boolean value (true or false) and may be used inside if statements:

```java
if (v1.isZeroLength())
if (v1.isParallelTo(v2))
if (v1.isPerpendicularTo(v2))
```

The vector class contains functions for the usual vector operations:

```java
len = v1.dotProduct(v2);
v3 = v1.crossProduct(v2);
```

The default constructor for a matrix initializes the matrix to the identity matrix:

```java
AcGeMatrix3d mat1, mat2, mat3;
```

The following rotates \( p_3 \) 90 degrees about the line defined by \( p_1 \) and \( v_1 \):

```java
mat1.setToRotation( kPi/2.0, v1, p1 );
p3 = mat1 * p2;
```

A matrix can be inverted if it is not singular:

```java
if (!mat2.isSingular())
    mat2.invert();
```

The * operator is defined for concatenating matrices:

```java
mat3 = mat1 * mat2;
```

The following tests if 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):

```java
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.

```cpp
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:

```cpp
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:

```cpp
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):

```cpp
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.

```cpp
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:

```cpp
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:

```cpp
if (line1.intersectWith(line2, p1))
if (line1.intersectWith(plane1, p1))
if (plane1.intersectWith(plane2, line1))
```
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 \( 2\pi \). Use these functions to determine whether a curve is periodic:

```cpp
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:

```cpp
Adesk::Boolean AcGeCurve2d::isClosed(const AcGeTol& = AcGeContext::gTol) const;
Adesk::Boolean AcGeCurve3d::isClosed(const AcGeTol& = AcGeContext::gTol) const;
```

A 3D curve can be planar, which means that all of its points reside in the same plane, or nonplanar. Use this function to determine whether a 3D curve is planar:

```cpp
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:

```cpp
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 modeling-space point that corresponds to a given parametric value. If your application performs evaluation frequently, you'll probably find the `AcGePointOnCurve2d` and `AcGePointOnCurve3d` classes more efficient (see “Special Evaluation Classes” on page 374). The curve functions for evaluating points are as follows:

```cpp
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 semantically, the object 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 the object in a nondegenerate form; in the previous example, it would return a point.

```cpp
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 is defined by its normal vectors. A parametric surface has two parameters, u and v, each of which defines 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 may reverse the orientation of a surface by calling the following function:

```cpp
AcGeSurface
AcGeSurface::reverseNormal()
```

If this function is called, then the surface evaluator returns the negative of the natural normal. The following function returns a value of true if the orientation of the surface is opposite to the natural orientation:

```cpp
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.

```cpp
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 of the closest point are obtained.

```cpp
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);

kntrs.append (0.0);
kntrs.append (0.0);
kntrs.append (0.0);
kntrs.append (1.0);
kntrs.append (1.0);
kntrs.append (1.0);
kntrs.append (1.0);
kntrs.append (1.0);

AcGePoint3dArray cntrlPnts;
AcGePoint3dArray cntrlPnts;
AcGePoint3dArray cntrlPnts;
AcGePoint3dArray cntrlPnts;

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 therefore 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.

Almost all 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.

The AcGe library contains evaluator functions for every curve and surface class, which may be called through the evalPoint() methods of the curve and surface classes. The curve and surface evaluators may also be accessed through the AcGePointOnCurve2d, AcGePointOnCurve3d, and AcGePointOnSurface classes. These classes serve two main purposes:

- They encapsulate all of 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, more natural, 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`:

```cpp
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:

```cpp
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 of 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:

```cpp
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:

```cpp
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`:

```cpp
// Compute closest point on 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()` need 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.

```cpp
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:

```cpp
AcGePoint3d  pnt3d = pntOnSrf.point (surf, param);
AcGeVector3d  uFirstPartial = pntOnSrf.uDeriv();
AcGeVector3d  vFirstPartial = pntOnSrf.vDeriv();
```

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, but they also provide a simpler and more natural interface to the curve and surface evaluators than the traditional evaluators of most geometric modeling systems. A typical C-style surface evaluator looks something like the following:

```cpp
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 you are 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 and natural fashion using the `point()`, `uDeriv()`, `vDeriv()`, `mixedPartial()`, and `normal()` functions. The names of these functions indicate clearly what 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 of most geometric modeling systems. 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 area cannot be part of the curve or surface classes because more than one user might be evaluating on the curve or surface simultaneously, which would mean that multiple users would be trying to use the same data area.

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 to actually 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:

```cpp
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`:

```cpp
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:

```cpp
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. It is a misuse of these classes to store large arrays of these objects or to have many of these objects in scope at one time that all point to the same curve or surface. These classes were designed to encapsulate the parametric and model space information of a point on a curve or surface and to provide a simple and efficient interface to the curve and surface evaluators.