Basic Interaction with AutoCAD

Topics in this section

- Basic Interaction with AutoCAD

Basic Interaction with AutoCAD

Topics in this section

- MFC Support in ObjectARX
- Selection Set, Entity, and Symbol Table Functions
- Plot API
- Global Functions for Interacting with AutoCAD
- ObjectARX Global Utility Functions
- AutoCAD Command Prompt Standard

MFC Support in ObjectARX

The Microsoft ® Foundation Class (MFC) library allows a developer to implement standard user interfaces quickly. ObjectARX ® applications can be created to take advantage of the MFC library. The ObjectARX environment provides a set of classes that a developer can use to create MFC-based user interfaces that behave and appear as the built-in Autodesk user interfaces. This section discusses how to use the MFC library as part of an ObjectARX application and how the AutoCAD ® built-in MFC system can be used to create dialogs that behave and operate like AutoCAD.

Topics in this section

- Using MFC with ObjectARX Applications
- ObjectARX Applications with Dynamically Linked MFC
- Built-In MFC User Interface Support
- Using AdUi and AcUi

Using MFC with ObjectARX Applications

ObjectARX supports dynamic MFC linking. Using an extension DLL is also recommended. In order to use the Autodesk AdUi and AcUi MFC base classes, you *must* dynamically link your MFC ObjectARX application and make it an extension DLL.

Note Linking to the static MFC library is not supported because some of the ObjectARX libraries link with the dynamic MFC library. A DLL cannot link to both the static and dynamic MFC libraries; linker warnings will result.

For complete information about MFC, see the Microsoft online help and technical notes. In particular, see TN011 and TN033 for information about using MFC as part of a DLL, which is an important concept for ObjectARX.

Topics in this section

- MFC and Modeless Dialog Boxes

MFC and Modeless Dialog Boxes

Since AutoCAD attempts to take focus away from all of its child windows, modeless dialogs have a special requirement. At regular intervals, the modeless dialog will get a `WM_ACAD_KEEPFOCUS` window message, which is defined in *adscodes.h* as 1001. When your dialog gets this message, it must return `TRUE` if it should keep focus. If the response to this message is `FALSE` (which is also the default), then your dialog box will lose focus as soon as the user moves the mouse pointer off the dialog box's window.

You can do this with the dialog box's message map, and an `ON_MESSAGE()` declaration such as

```
BEGIN_MESSAGE_MAP(HelloDlg, CDialog)
    ON_COMMAND(IDCLOSE, OnClose)
    ON_COMMAND(IDC_DRAW_CIRCLE, OnDrawCircle)
    ON_MESSAGE(WM_ACAD_KEEPFOCUS, onAcadKeepFocus)
END_MESSAGE_MAP()
```

In this example, the application's dialog class is `HelloDlg`, which is derived from `CDialog`. When you add this entry to the message map, you must also write a handler function for the message. Assume you have written a function called `keepTheFocus()`, which returns `TRUE` if your dialog wants to keep the input focus and `FALSE` if the dialog is willing to yield the focus to AutoCAD. An example message handler is provided here:

```
afx_msg LONG HelloDlg::onAcadKeepFocus(UINT, LONG)
{
    return keepTheFocus() ? TRUE : FALSE;
}
```

ObjectARX Applications with Dynamically Linked MFC

The supported method for building an MFC-based ObjectARX application is to use the dynamically linked MFC libraries.

To build an ObjectARX application using the shared MFC library

1. Select the MFC DLL option for the project.
2. In the Application Settings, select MFC extension DLL.
3. Add an `acrxEntryPoint` function to the project's CPP file. See the example at the end of the section for a complete setup for an MFC project.

Topics in this section

- Debugging ObjectARX Applications with Dynamic MFC
- Resource Management

Debugging ObjectARX Applications with Dynamic MFC

When debugging ObjectARX applications built with a dynamically linked MFC library, link with the release version of C runtime and MFC libraries. This allows use of the MFC or C runtime debugging facilities, but does not allow stepping into the Microsoft MFC debugging source code.

Resource Management

Resource management is an important consideration when designing an ObjectARX application that uses an MFC library shared with AutoCAD and other applications.

You must insert your module state (using `CDynLinkLibrary`) into the chain that MFC examines when it performs operations such as locating a resource. However, it is strongly recommended that you explicitly manage your application's resources so that they will not conflict with other resources from AutoCAD or other ObjectARX applications.

To explicitly set resources

1. Before taking any steps that would cause MFC to look for your resource, call the AFX function `AfxSetResourceHandle()` to set the custom resource as the system default.
2. Before setting the system resource to your resource, call `AfxGetResourceHandle()` to get and store the current system resource handle.
3. Immediately after performing any functions that require the custom resource, the system resource should be reset to the resource handle previously saved.

When calling AutoCAD API functions or invoking AutoCAD commands that need AutoCAD's resources, such as `acedGetFileD()`, be sure to set the resource back to AutoCAD before making the function call. Restore your application's resource afterwards. Use `acedGetAcadResourceInstance()` to get AutoCAD's resource handle.

Topics in this section

- CAcExtensionModule Class
- CAcModuleResourceOverride Class

CAcExtensionModule Class

The ObjectARX SDK provides two simple C++ classes that can be used to make resource management easier. The `CAcExtensionModule` class serves two purposes—it provides a placeholder for an `AFX_EXTENSION_MODULE` structure (normally used to initialize or terminate an MFC extension DLL) and tracks two resource providers for the DLL. The resource providers are the module's resources (which are normally the DLL itself, but may be set to some other module) and the default resources (normally the host application, but are actually the provider currently active when `AttachInstance()` is called). `CAcExtensionModule` tracks these to simplify switching MFC resource lookup between the default and the module's. A DLL should create one instance of this class and provide the implementation for the class.

CAcModuleResourceOverride Class

Use an instance of this class to switch between resource providers. When the object is constructed, a new resource provider will get switched in. Upon destruction, the original resource provider will be restored. The following code provides an example:

```
void MyFunc ()
{
    CAcModuleResourceOverride myResources;
}
```

Upon entry to this function the module's resources will be selected. When the function returns, the default resources will be restored. A resource override can be selected in any of the following ways:

- Use the default constructor (no arguments), or pass `NULL` (or 0) to the constructor. The DLL's resources will be selected. The default resources will be restored when the `CAcModuleResourceOverride` destructor is called. The DLL and default resource handles are tracked by the DLL's `CAcExtensionModule`.
- Pass a non-`NULL` handle to the constructor. The resources of the module associated with the given handle will be selected. The default resources will be restored when the `CAcModuleResourceOverride` destructor is called.

Two macros—`AC_DECLARE_EXTENSION_MODULE` and `AC_IMPLEMENT_EXTENSION_MODULE`—help define and implement the classes in your application.

The following code illustrates how to make use of the `CAcExtensionModule` and `CAcModuleResourceOverride` classes in an ObjectARX application:

```
AC_IMPLEMENT_EXTENSION_MODULE(theArxDLL);
HINSTANCE _hdllInstance = NULL;
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    // Remove this if you use lpReserved
    UNREFERENCED_PARAMETER(lpReserved);
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        theArxDLL.AttachInstance(hInstance);
        hdllInstance = hInstance;
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        theArxDLL.DetachInstance();
    }
    return 1;   // ok
}
```

Built-In MFC User Interface Support

ObjectARX has a set of MFC User Interface (UI)-related classes that easily allow you to provide a consistent UI. This means your UI can behave like and have the appearance of the AutoCAD UI. It is highly recommended that you use these classes because they allow your application to be more tightly integrated with the AutoCAD UI. The Autodesk MFC extension classes include the AdUi and AcUi libraries, as well as the libraries that support features like file navigation dialogs and tool palettes. AdUi is not AutoCAD-specific. AcUi contains AutoCAD-specific appearance and behavior. For a complete MFC extension class hierarchy diagram, see *classmap.dwg* in the ObjectARX *classmap* directory.

AdUi is an MFC extension DLL used to extend some of the UI-related classes of MFC. The library was developed for use with AutoCAD and other Autodesk products and contains core functionality. The companion library, AcUi, builds upon the AdUi framework and provides AutoCAD-specific appearance and behavior. The AdUi and AcUi libraries provide classes that extend those provided by MFC in ways that allow ObjectARX developers to use the same UI functionality found in AutoCAD. MFC developers can seamlessly use these classes. Listed below are some of the main areas of added functionality provided by AdUi and AcUi.

To use AdUi in an MFC-based application, the project's C++ source files must include *adui.h* and the project should link *adui19.lib* (the *adui19.dll* import library).

To use AcUi in an MFC-based AutoCAD application, the project's C++ source files must include *adui.h*, then *acui.h*, and the project should link *acui19.lib* and *adui19.lib*. AutoCAD invokes the library's initialization routine, `InitAcUiDLL()`, which also handles the AdUi initialization (via an `InitAdUiDLL()` call); therefore your application need not reinitialize AcUi or AdUi.

Warning Although *adui19.dll* may be called from MFC-based applications other than AutoCAD (or other Autodesk products), the library's intended use is by Autodesk and third parties expressly for the creation of software to work exclusively with AutoCAD, or other Autodesk products. Use of this DLL for non-AutoCAD, standalone products is not permitted under the AutoCAD license agreement.

AdUi and AcUi provide classes that implement features such as the following:

- Dialog resizing
- Dialog data persistency
- Tabbed dialogs
- Extensible tabbed dialogs
- Context-sensitive help and F1 help
- Dialog interaction with AutoCAD's drawing editor
- Bitmap buttons that are easy to use
- Static bitmap buttons
- Bitmap buttons that are drag and drop sites
- Toolbar-style bitmap buttons
- Owner-draw buttons that are easy to use
- Dialog and control support for standard ToolTips
- Dialog and control support for TextTips (which display truncated text)
- Dialog and control support for DrawTips (owner-draw TextTips)
- Combo boxes that display and allow the selection of many AutoCAD specific items
- Docking control bar windows for use with AutoCAD
- AutoCAD-specific bitmap buttons (stock Pick and Select buttons)
- Specialized edit controls that can perform AutoCAD-specific data validation
- Custom messaging, including data validation

Note If you include either *acui.h* or *acuinavdialog.h* in your source, you must include the *afxole.h* header file beforehand. If *afxole.h* is not included first, multiple compiler errors will result.

Topics in this section

- AcUi Button Classes
- AcUi Dialog Classes
- AcUi MRU Combo Boxes
- AdUi Messaging
- AdUi Tip Windows
- AdUi Dialog Classes
- AdUi Classes Supporting Tab Extensibility
- AdUi and AcUi Control Bar Classes
- AdUi and AcUi Edit Controls
- AdUi and AcUi Combo Box Controls
- AdUi Button Classes
- Dialog Data Persistency
- Using and Extending the AdUi Tab Dialog System
- Constructing a Custom Tab Dialog that is Extensible

- Extending the AutoCAD Built-in Tab Dialogs
- Registering Objects Derived from CAdUiDockControlBar

AcUi Button Classes

These controls build upon the AdUi classes and are usable only with AutoCAD.

CAcUiPickButton Class

`CAcUiPickButton` specializes `CAcUiBitmapButton`, which is a wrapper for the class `CAdUiBitmapButton`. `CAcUiPickButton` provides a button that displays a standard pick button bitmap.

CAcUiSelectButton Class

`CAcUiSelectButton` specializes `CAcUiPickButton`. It provides a button that displays a standard selection button bitmap.

AcUi Dialog Classes

The AcUi dialog classes build upon the AdUi dialog classes and are usable only with AutoCAD.

CAcUiDialog Class

`CAcUiDialog` is a general-purpose class that provides a set of member functions allowing for resizable dialogs and data persistency in AutoCAD.

CAcUiTabMainDialog Class

`CAcUiTabMainDialog` represents the main container dialog in an AutoCAD tabbed dialog. `CAcUiTabMainDialog` and `CAcUiTabMainDialog` are used in place of `CPropertySheet` and `CPropertyPage` to construct tabbed dialogs in AutoCAD.

CAcUiTabChildDialog Class

`CAcUiTabChildDialog` represents a tab in a tabbed dialog. `CAcUiTabMainDialog` and `CAcUiTabChildDialog` are used in place of `CPropertySheet` and `CPropertyPage` to construct tabbed dialogs in AutoCAD. Each tab in an AutoCAD tabbed dialog is a `CAcUiTabChildDialog`.

CAcUiAlertDialog Class

`CAdUiAlertDialog` represents an alert dialog with three buttons. One button is the CANCEL button and the other two button labels are set by the programmer. It is a general-purpose alert dialog.

CAcUiFileDialog Class

`CAcUiFileDialog` provides an AutoCAD-specific derivation of `CAdUiFileDialog`.

AcUi MRU Combo Boxes

AcUi extends combo box support to manage an MRU (most recently used) list automatically within the control. The basic functionality is provided by the class `CAcUiMRUComboBox` (derived from `CAcUiComboBox`). A companion class, `CAcUiMRUListBox`, provides DrawTip support for the combo box's `ComboLBox`. This is necessary due to the MRU combo box implementation as an owner-draw control.

Five specialized MRU combo box classes are also provided: `CAcUiArrowHeadComboBox`, `CAcUiColorComboBox`, `CAcUiLineWeightComboBox`, `CAcUiPlotStyleTablesComboBox`, and `CAcUiPlotStyleNamesComboBox`. These provide standard user interfaces for managing dimensioning arrowheads, color and lineweight selections, and plot style table and plot style names selection.

CAcUiMRUComboBox Class

`CAcUiMRUComboBox` inherits `CAcUiComboBox` and serves as the base class for owner-draw combo boxes that implement an MRU list. Each item in the list can contain a small image followed by some text. Each item also tracks a unique value, referred to as cargo, and maintained as standard Windows ® ITEMDATA within the control. The class features built-in support for up to two generic, optional items, referred to as Option1 and Option2. These usually correspond to "ByLayer" and "ByBlock" and often have special significance. Two other items, Other1 and Other2, may also be enabled and appear only when the list is dropped down. Selecting either of these items triggers a special event within the control.

CAcUiArrowHeadComboBox Class

`CAcUiArrowHeadComboBox` specializes `CAcUiMRUComboBox` for dimensioning arrowhead selection. The control displays bitmaps representing the standard AutoCAD dimensioning arrowhead styles, which are always present in the list. By default no optional or additional items are present or added. The cargo associated with each item is the AutoCAD index for the associated stock arrowhead. When MRU items are added to the list, they are automatically assigned a unique cargo value (which will be greater than the AutoCAD index for a user-defined arrowhead style).

CAcUiColorComboBox Class

`CAcUiColorComboBox` specializes `CAcUiMRUComboBox` for color selection. The control displays color swatches representing selections from AutoCAD's palette. The stock items always present in the control reflect color numbers 1 through 7. Both optional items are used; Option1 displays "ByLayer" and Option2 displays "ByBlock". MRU items display "Color nnn," where nnn is the associated color number. The cargo associated with each item indicates an AutoCAD color number (such as 1 to 255), "ByBlock" relates to 0, and "ByLayer" corresponds to 256. The Other1 item is enabled and triggers the AutoCAD Color Selection dialog. If Other2 is enabled it displays as "Windows..." and by default triggers the Windows Color Selection Common dialog. If the user selects an item from either of these dialogs the selection appears in the MRU list and becomes the current item in the control.

CAcUiLineWeightComboBox Class

`CAcUiLineWeightComboBox` specializes `CAcUiMRUComboBox` for lineweight selection. The control displays a small preview of the lineweights AutoCAD supports, ranging from 0.05mm to 2.11mm, and includes "None" and optionally "Default". Both metric and imperial values are displayed, depending on the setting of the LWUNITS system variable. Both optional items are used; Option1 displays "ByLayer" and Option2 displays "ByBlock". Each item maintains cargo that corresponds to the item's `AcDb::kLnWtxxx` value.

CAcUiPlotStyleTablesComboBox Class

CAcUiPlotStyleTablesComboBox specializes CAcUiMRUComboBox for plot style table selection. The control displays plot style table names according to the current plot style mode (color-dependent mode or named plot styles). The MRU functionality of the combo box is not used. A bitmap indicating an embedded translation table is displayed in named plot style mode for those tables that have an embedded translation table.

CAcUiPlotStyleNamesComboBox Class

CAcUiPlotStyleNamesComboBox specializes CAcUiMRUComboBox for plot style name selection. The MRU functionality of the combo is not used, and "ByLayer", "ByBlock", and "Other..." items can be conditionally displayed. If present, the "Other..." item can trigger either the Assign Plot Style dialog or the Set Current Plot Style dialog.

CAcUiMRUListBox Class

CAcUiMRUListBox derives from CAcUiListBox. It is used by CAcUiMRUComboBox to subclass the control's list box (ComboLBox) and provide DrawTip support. Advanced applications that use specialized MRU combo boxes may need to derive special MRU list boxes to display DrawTips correctly.

CAcUiTrueColorComboBox Class

CAcUiTrueColorComboBox specializes CAcUiMRUComboBox for color selection. The color combo box displays color swatches representing the colors given to it as AcCmColor objects.

AdUi Messaging

The AdUi library uses an internal messaging scheme to facilitate communication between objects. Typically this involves a container (such as a dialog) responding to a notification from a contained window (such as a control). Advanced applications may tailor the built-in system to their needs, or add AdUi messaging support to other CWnd derived classes.

AdUi Tip Windows

AdUi provides three types of tip windows: ToolTips, TextTips, and DrawTips. ToolTips represent stock Windows ToolTips, as provided by the Common Controls DLL installed on the user's system. TextTips are text-based tip windows that pop up over a control, usually to reveal data that the user would otherwise have to scroll into view. DrawTips are an extension of TextTips. The control underneath the tip is usually responsible for painting the contents of the tip (analogous to an owner-draw tip).

Most applications rarely involve these classes directly, since AdUi usually handles all of the requirements. AdUi uses its internal messaging system to negotiate between containers and controls and decide when and how to display a tip.

CAdUiTipWindow Class

CAdUiTipWindow is the basic AdUi tip window class. These objects handle generic tip display and know when to automatically hide themselves (such as detecting cursor

movement, a brief time-out, or keyboard activity).

CAdUiTextTip Class

`CAdUiTextTip` specializes `CAdUiTipWindow` to display a TextTip.

CAdUiDrawTipText Class

`CAdUiDrawTipText` is used internally by the AdUi messaging system to inform a control that a tip window needs repainting. The control has the option of changing attributes of the tip window's device context and drawing the text.

AdUi Dialog Classes

The AdUi dialog classes are usable in applications other than AutoCAD.

CAdUiBaseDialog Class

`CAdUiBaseDialog` provides basic support for tip windows (ToolTips and TextTips) and the AdUi message handling system. It also supports context help and F1 help in dialogs. It is the common base class for all dialogs except those based on the common file dialog.

CAdUiDialog Class

`CAdUiDialog` is a general purpose class that provides a set of member functions allowing for resizable dialogs and data persistency.

CAdUiFileDialog Class

`CAdUiFileDialog` specializes `CFileDialog` much the same way as `CAdUiBaseDialog` specializes `CDialog`. The class provides basic support for tip windows (ToolTips and TextTips), context help and AdUi message handling in a common file dialog. Unlike `CAdUiBaseDialog`, there is no built-in support for position and size persistency.

CAdUiHideableDialogSettings

`CAdUiHideableDialogSettings` contains settings for hideable dialogs. By default these dialogs do not display, though through the `setState()` method this can be changed.

CAdUiTabMainDialog Class

`CAdUiTabMainDialog` represents the main container dialog in a tabbed dialog. `CAdUiTabMainDialog` and `CAdUiTabMainDialog` are used in place of `CPropertySheet` and `CPropertyPage` to construct tabbed dialogs.

CAdUiTabChildDialog Class

`CAdUiTabChildDialog` represents a tab in a tabbed dialog. `CAdUiTabMainDialog` and `CAdUiTabChildDialog` are used in place of `CPropertySheet` and `CPropertyPage` to construct tabbed dialogs. Each tab in a tabbed dialog is a `CAdUiTabChildDialog`.

AdUi Classes Supporting Tab Extensibility

The following classes provide support for tab dialogs.

CAdUiTabExtensionManager Class

`CAdUiTabExtensionManager` is a class that manages adding and removing tabs from a tabbed dialog that is extensible. If a dialog is tab extensible, an instance of this class is found in the `CAdUiTabMainDialog`.

CAdUiTab Class

`CAdUiTab` encapsulates the MFC `CTabCtrl` and adds functionality to it. One of these objects is found in the main dialog object.

AdUi and AcUi Control Bar Classes

The following classes provide support for docking windows.

CAdUiDockControlBar Class

The `CAdUiDockControlBar` class, part of a docking system, adds extended capabilities to the MFC `CControlBar` class. The main feature provided is the resizing of the control bars when docked. More than one control bar can be docked together, each of them being able to be resized individually using splitters created by the docking system. `CAdUiDockControlBar` also comes with a gripper bar and a close button when docked. Control bars' state can be switched from docked to undocked or vice versa, by double-clicking on the gripper when docked, or the title bar when undocked, or by dragging them with the mouse. The docking system handles the persistency of the control bars, preserving their position and state across sessions. Finally, `CAdUiDockControlBar` provides a default context menu to control the bar behavior, with a possibility for the developer to customize this menu.

CAcUiDockControlBar Class

The `CAcUiDockControlBar` class adds to the `CAdUiDockControlBar` class a behavior common to AutoCAD dockable tools: when the user moves the mouse cursor out of the control bar region, the focus is automatically given back to AutoCAD.

AdUi and AcUi Edit Controls

The following classes provide specialized editing controls, including support for specific types of data.

CAdUiEdit Class

`CAdUiEdit` is derived from the `CEdit` class to provide edit box controls. This class provides support for tip windows for truncated text items (TextTips). This class takes bit flags to add desired validation behavior, based on the following types of input: Numeric, String, Angular, and Symbol names. Generally you should use one of the classes derived from the AutoCAD-specific class `CAcUiComboBox`, which adds a specific data type validation and persistency to the control. These are `CAcUiStringEdit`, `CAcUiSymbolEdit`, `CAcUiNumericEdit`, and `CAcUiAngleEdit`.

CAcUiEdit Class

CAcUiEdit provides an AutoCAD-specific derivation of CAdUiEdit.

CAcUiAngleEdit Class

CAcUiAngleEdit is derived from CAcUiEdit and provides a specialized constructor to ensure that the AC_ES_ANGLE style bit is always set in the style mask. Objects of this class are intended for use in editing angular/rotational data specific to AutoCAD settings.

CAcUiNumericEdit Class

CAcUiNumericEdit is derived from CAcUiEdit and provides a specialized constructor to ensure that the AC_ES_NUMERIC style bit is always set in the style mask. Objects of this class are intended for use in editing numeric data (such as distance) specific to AutoCAD settings.

CAcUiStringEdit Class

CAcUiStringEdit is derived from CAcUiEdit and provides a specialized constructor to ensure that the AC_ES_STRING style bit is always set in the style mask. Any input is acceptable.

CAcUiSymbolEdit Class

CAcUiSymbolEdit is derived from CAcUiEdit and provides a specialized constructor to ensure that the AC_ES_SYMBOL style bit is always set in the style mask. Objects of this class are intended for use in editing valid AutoCAD symbol names.

CAdUiListBox Class

CAdUiListBox specializes the MFC CListBox to provide a control that supports AdUi messaging. The class can be used anywhere a CListBox can be used. Since it provides the additional container-side support for AdUi registered messages, it is convenient to use CAdUiBaseDialog (or a derived class) with the CAdUiListBox (or a derived class) controls.

CAdUiListBox provides features that allow the class to be used to subclass a list box included in a combo box. When used in concert with a CAdUiComboBox, the list box is able to track the combo box and, in the case of an owner-draw control, either delegate drawing to the combo box or provide its own drawing routines.

CAdUiListCtrl Class

CAdUiListCtrl is derived from CListCtrl class to provide list controls. This class provides support for tip windows for truncated text items (TextTips). TextTips will appear for truncated header items for list controls in a report view, and for individual truncated text items in columns in the body of a list control. Owner-drawn controls are supported.

CAdUiHeaderCtrl Class

CAdUiHeaderCtrl specializes CHeaderCtrl. Most often, CAdUiHeaderCtrl represents the subclassed header contained in a list control (CAdUiListCtrl). You do not need to subclass the header control to get TextTip support for column headers in a list control (provided automatically in CAdUiListCtrl).

AdUi and AcUi Combo Box Controls

The following classes provide support for combo box controls.

CAdUiComboBox Class

CAdUiComboBox is derived from the CComboBox class to provide combo box controls. This class provides support for tip windows for truncated text items (TextTips), and data validation in the edit control. This class takes bit flags to add desired validation behavior, based on the following types of input: numeric, string, angular, and symbol names. Generally, you should use one of the classes derived from the AutoCAD-specific class CAcUiComboBox, which adds a specific data type validation and persistency to the control. These are CAcUiStringComboBox, CAcUiSymbolComboBox, CAcUiNumericComboBox, and CAcUiAngleComboBox. Support for owner-drawn controls is also built in.

CAcUiAngleComboBox Class

The CAcUiAngleComboBox constructor automatically creates a CAcUiAngleEdit to subclass the control's edit box. This allows for validation of angles specific to AutoCAD settings.

CAcUiNumericComboBox Class

The CAcUiAngleComboBox constructor automatically creates a CAcUiNumericEdit to subclass the control's edit box. This allows for validation of numbers specific to AutoCAD settings.

CAcUiStringComboBox Class

The CAcUiStringComboBox constructor automatically creates a CAcUiStringEdit to subclass the control's edit box. Any input is acceptable.

CAcUiSymbolComboBox Class

The CAcUiSymbolComboBox constructor automatically creates a CAcUiSymbolEdit to subclass the control's edit box. Valid AutoCAD symbol names are acceptable input.

AdUi Button Classes

These controls are usable in applications other than AutoCAD.

CAdUiOwnerDrawButton Class

This class provides a basic owner-draw button. The class can be used anywhere a CButton can be used. When used in an AdUi-derived dialog (or a class that supports AdUi messaging) CAdUiOwnerDrawButton automatically provides for the display of an AdUi tip window. The class also supports drag and drop, Static and Tool Display, and PointedAt effects. In Tool Display mode, the button appears flat and pops up when pointed at (such as when the mouse moves over the button). Clicking the button makes it push down. In Static Display mode, the button appears flat and behaves more like a static control than a push button. The combination of enabling drag and drop and Static Display is appropriate for creating sites that receive files via drag and drop.

CAdUiBitmapButton Class

This class specializes CAdUiOwnerDrawButton to provide a button that displays a bitmap (the image is drawn transparently in the button). By default, objects of this class automatically resize to fit the associated bitmap image. Unlike MFC's CBitmapButton, only one bitmap is needed to define all of the button states (MFC's

class requires four bitmaps).

CAdUiBitmapStatic Class

`CAdUiBitmapStatic` specializes `CAdUiBitmapButton` to provide a button that enables Static Display by default. These controls act more like statics than pushbuttons.

CAdUiDropSite Class

`CAdUiDropSite` specializes `CAdUiBitmapStatic` to provide a button that enables drag and drop as well as Static Display. These controls can receive files via drag and drop.

CAdUiToolButton Class

`CAdUiToolButton` specializes `CAdUiBitmapButton` to provide a button that enables Tool Display by default. These controls appear more like toolbar buttons than regular pushbuttons.

Dialog Data Persistency

`CAcUiDialog` and the `CAcUiTab` classes automatically inherit persistency. Persistency, as defined by the dialogs and controls in *AcUi19.dll*, means that storage for any and all user modal dialogs in AutoCAD derived from these classes will store data with the current user profile, making it a virtual preference.

Your dialog should have a unique name because it will use a shared area of the user profile registry space. Given that developers usually create their applications using their registered developer prefix, the following method is recommended:

module-name:dialog-name

For example, if your ObjectARX application is named AsdkSample and you have a dialog titled Coordinates, you would name it AsdkSample:Coordinates. For more information, see SetDialogName.

There are two types of dialog data persistency: out-of-the-box and developer-defined. Out-of-the-box persistency refers to dialog position, size, and list view column sizes. Developer-defined refers to any data that a developer chooses to store in the user profile either during the lifetime or dismissal of the dialog and which may be retrieved across dialog invocations.

Using and Extending the AdUi Tab Dialog System

All tabbed dialogs that use `CAdUiTabMainDialog` and `CAdUiTabChildDialog` can be easily made tab extensible. There is no limit for the number of tabs that can be added to a tab-extensible dialog. If the main dialog is resizable, added tabs can participate in that resizing using the same directives outlined in the documentation on resizable dialogs. All dialogs in AutoCAD use scrolling tabs as opposed to stacked tabs.

It is important for you to set the dirty bit for the extended tab using the `SetDirty()` member function of `CAdUiTabChildDialog` when data needs to be initialized or updated via `DoDataExchange`.

Constructing a Custom Tab Dialog that is Extensible

Construct your tabbed dialog using CAcUiTabMainDialog for the main dialog frame and CAcUiTabChildDialog for each tab. In the OnInitDialog() or constructor of the CAcUiTabMainDialog immediately call SetDialogName() with the published name of your extensible dialog. ObjectARX applications will use this name to add tabs to your dialog. After you add your tabs with calls to AddTab(), in OnInitDialog, call AddExtendedTabs(). Remember that your tabbed dialog can have any number of added tabs in it, so do not assume a fixed number of tabs elsewhere in the dialog's code.

For example:

```
BOOL CPrefTabFrame::OnInitDialog()
// Dialog initialization for my tabbed dialog frame.
{
    SetDialogName("Preferences");
    CAcUiTabMainDialog::OnInitDialog();
    ...
    // Add my tabs here.
    m_tab.AddTab(0,IDS_FILES_TABNAME,IDD_FILES_TAB,&m_filesTab);
    m_tab.AddTab(1,IDS_PERF_TABNAME,IDD_PERF_TAB,&m_performTab);
    m_tab.AddTab(2,IDS_COMP_TABNAME,IDD_COMP_TAB,&m_compatTab);
    // Add any extended tabs. This call is what makes this
    // dialog tab extensible
    AddExtendedTabs();
}
```

Extending the AutoCAD Built-in Tab Dialogs

Use Class Wizard or some other means to create your tab subclassed from CDialog. In the properties for the dialog, change the style of the dialog to “child” and the border to “resizing”. Implement an override for PostNcDestroy(). Replace all occurrences of CDialog with CAdUiTabExtension in all source files for the dialog. In PostNcDestroy() for the tab extension delete the tab object that has been allocated (see example below).

In your AcRx::kInitAppMsg handler in acrxEntryPoint() add a call to acedRegisterExtendedTab("MYAPPNAME.ARX", "DIALOGNAME"), where MYAPPNAME is the base file name of your application and DIALOGNAME is the published name of the extensible tabbed dialog you wish to add to.

Implement an AcRx::kInitDialogMsg handler in acrxEntryPoint() and add the tab there. The (void*)appId argument to acrxEntryPoint() is a CAdUiTabExtensionManager pointer. Use the member function GetDialogName() for the CAdUiTabExtensionManager to get the name of the dialog being initialized and, if the application wants to add to this dialog, call the AddTab() member function of the CAdUiTabExtensionManager to add the tab. If the dialog is resizable and you want some of your controls to resize, add that resizing code after the call to AddTab().

For example:

```
extern "C" AcRx::AppRetCode acrxEntryPoint(
    AcRx::AppMsgCode msg, void* appId)
{
    switch (msg) {
    case AcRx::kInitAppMsg:
        acrxDynamicLinker->unlockApplication(appId);
        acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
        break;
    case AcRx::kInitDialogMsg:
    // A dialog is initializing that we are interested in adding
    // tabs to.
        addMyTabs((CAdUiTabExtensionManager*)pkt);
        break;
    default:
        break;
    }
    return AcRx::kRetOK;
}
void initApp()
{
    InitMFC();
    // Do other initialization tasks here.
    acedRegCmds->addCommand(
        "MYARXAPP",
        "MYARXAPP",
        "MYARXAPP",
        ACRX_CMD_MODAL,
        &MyArxAppCreate);
    // Here is where we register the fact that we want to add
    // a tab to the Options dialog.
    acedRegisterExtendedTab("MYARXAPP.ARX", "OptionsDialog");
}
// CMyTab1 is subclassed from CAcUiTabExtension.
static CMyTab1* pTab1;
void addMyTabs(CAdUiTabExtensionManager* pXtabManager)
{
    // Allocate an extended tab if it has not been done already
    // and add it through the CAdUiTabExtensionManager.
    pTab1 = new CMyTab1;
    pXtabManager->AddTab(_hdllInstance, IDD_TAB1,
        "My Tab1", pTab1);
    // If the main dialog is resizable, add your control
    // resizing directives here.
    pTab1->StretchControlXY(IDC_EDIT1, 100, 100);
}
```

Then, for the `CMyTab1` class implementation:

```
void CMyTab1::PostNcDestroy()
// Override to delete added tab.
{
    delete pTab1;
    pTab1 = NULL;
    CAcUiTabExtension::PostNcDestroy();
}
```

Registering Objects Derived from CAdUiDockControlBar

Windows derived from `CAdUiDockControlBar`—which includes `CAcUiDockControlBar`, `CAdUiPaletteSet`, and `CAcTcUiToolPaletteSet`—can register themselves using the global function `AdUiRegisterTool()`. Registered windows that were visible when a session of AutoCAD closed will be automatically started in the next session and will be saved and restored in workspaces.

In order for the window to appear in the Customize User Interface dialog (CUI dialog) and to operate in workspaces more safely, an entry for it should also be added to the system registry.

Add the following key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R20.0\
    ACAD-E001:409\DockingTools\<classId>
```

The class ID is returned by the `CAdUiDockControlBar::GetToolID()` method, and it can be turned into a string using the COM method `StringFromCLSID()`. For example, the class ID string for Properties palette is `{6D32A2D9-832E-11D2-A83C-0060B0872C0B}`.

This key should contain the following properties:

Property Name	Type	Description of Data
Command	REG_SZ	Command to invoke the window
CommandClose	REG_SZ	Command to close the window
Name	REG_SZ	Localized name of the window

The two command names should be the global command names, but without any preceding underscore. For example, the global command for the Properties palette is "PROPERTIES", not "_PROPERTIES". The command to invoke the window should be the same command registered using `AdUiRegisterTool()`. The requirement for a command to close the window was introduced in AutoCAD 2006, so your window might not have such a command. As a fallback, AutoCAD will attempt to close the window using direct Windows messages when necessary.

The Name property should be a localized descriptive name of the window. This is the name by which the CUI dialog will refer to the window when populating a workspace.

You can test whether the registry entries are correct by checking whether the window is listed in the CUI dialog and whether the window can be opened and closed by switching to and from a workspace that contains it.

Using AdUi and AcUi

The example in this section shows how to use AdUi and AcUi to create a dialog box. The source code for this example can be found in the ObjectARX *samples\editor\mfcsamps\acuisample_dg* directory. This documentation describes how to set up your project.

Topics in this section

- Create the ObjectARX MFC Application Skeleton
- Create the MFC Dialog
- Create the Class and Controls
- Create the Handlers for the Dialog
- Add Code to the Handlers

Create the ObjectARX MFC Application Skeleton

To create a project for an ObjectARX MFC application

1. Create a new MFC DLL project called *AsdkAcUiSample*.
2. In the Application Settings, select MFC extension DLL, and click Finish.
3. Open the generated CPP file. Remove the `AFX_EXTENSION_MODULE` call, add the `AC_IMPLEMENT_EXTENSION_MODULE` call, and revise the `DllMain()` function as described in the CAcModuleResourceOverride Class section.
4. Add the following code to set up the AutoCAD command and `acrxEntryPoint`:

```
void dialogCreate()
{
    acutPrintf("\nAcUi Dialog Sample");
}
```

The following `addCommand()` call uses the module resource instance from the `AC_IMPLEMENT_EXTENSION_MODULE` macro:

```
static void initApp()
{
    CAcModuleResourceOverride resOverride;
    acedRegCmds->addCommand(
        "ASDK_ACUI_SAMPLE",
        "ASDKACUISAMPLE",
        "ACUISAMPLE",
        ACRX_CMD_MODAL,
        dialogCreate,
        NULL,
        -1,
        theArxDLL.ModuleResourceInstance());
}
```

The following `unloadApp()` function is called when the application unloads. At this time it is important to detach the resource instance:

```
static void unloadApp()
{
    // Do other cleanup tasks here
    acedRegCmds->removeGroup("ASDK_ACUI_SAMPLE");
    theArxDLL.DetachInstance();
}
// Entry point
//
extern "C" AcRx::AppRetCode acrxEntryPoint(
    AcRx::AppMsgCode msg, void* appId)
{
    switch( msg )
    {
        case AcRx::kInitAppMsg:
            acrxDynamicLinker->unlockApplication(appId);
            acrxDynamicLinker->registerAppMDIAware(appId);
            initApp();
            break;
        case AcRx::kUnloadAppMsg:
            unloadApp();
            break;
        case AcRx::kInitDialogMsg:
            break;
        default:
            break;
    }
    return AcRx::kRetOK;
}
```

Create an *AsdkAcUiSample.h* header file and add the following lines to the file:

```
#include "resource.h" // main symbols
#define PI  3.14159265359
// Forward declaration for the entry point function of
// our application
void testCreate();
```

Add the following include files to *AsdkAcUiSample.cpp*:

```
#include "AsdkAcUiSample.h"
#include "AcExtensionModule.h"
```

You also need to add the ObjectARX libraries to the project file, change the *.dll* extension to *.arx*, and modify the *.def* file with the proper exports.

Then you can compile and load the application.

Create the MFC Dialog

To create an MFC dialog for an ObjectARX application

1. Add a dialog resource to your MFC application project.

2. Create the following dialog box using the controls:
3. Make sure the resource IDs match this diagram; otherwise, the remaining code will not work.

Create the Class and Controls

To create the class and controls associated with the MFC dialog

1. Add a new class for the dialog.
2. In the MFC Class Wizard, enter `AsdkAcUiDialogSample` for the dialog name, and click Finish.
3. Add the following member variables:
   - For the `IDC_BUTTON_ANGLE` and `IDC_BUTTON_POINT` resources, add `CButton` controls called `m_ctrlAngleButton` and `m_ctrlPickButton`, respectively.
   - For the `IDC_EDIT_ANGLE`, `IDC_EDIT_XPT`, `IDC_EDIT_YPT`, and `IDC_EDIT_ZPT` resources, add `CEdit` controls called `m_ctrlAngleEdit`, `m_ctrlXPtEdit`, `m_ctrlYPtEdit`, and `m_ctrlZPtEdit`, respectively.
   - For the `IDC_LIST_BLOCKS` resource, add a `CListBox` control called `m_ctrlBlockList`.
   - For the `IDC_COMBO_REGAPPS` resource, add a `CComboBox` control called `m_ctrlRegAppComboBox`.
4. Open the *AsdkAcUiDialogSample.h* header file and change the derivation of the new dialog class. It should be derived from `CAcUiDialog`:

```
class AsdkAcUiDialogSample : public CAcUiDialog
```

Change the types to use the AcUi controls. In *AsdkAcUiDialogSample.h*, change the control list to the following:

```
CAcUiSymbolComboBox   m_ctrlRegAppComboBox;
CAcUiListBox          m_ctrlBlockListBox;
CAcUiPickButton       m_ctrlPickButton;
CAcUiPickButton       m_ctrlAngleButton;
CAcUiAngleEdit        m_ctrlAngleEdit;
CAcUiNumericEdit      m_ctrlXPtEdit;
CAcUiNumericEdit      m_ctrlYPtEdit;
CAcUiNumericEdit      m_ctrlZPtEdit;
```

Add helper functions and member variables to track the point and angle values. These should be added to the public section of the class:

```
AcGePoint3d m_ptValue;
double m_dAngle;
void DisplayPoint();
bool ValidatePoint();
void DisplayAngle();
bool ValidateAngle();
void DisplayBlocks();
void DisplayRegApps();
```

Create the Handlers for the Dialog

To create handlers for the MFC dialog

1. Open the *AsdkAcUiDialogSample.cpp* source file in the editor.
2. At the top of the `AsdkAcUiDialogSample::OnInitDialog()` function definition, set the dialog's name, and then call the parent class version of `OnInitDialog()`:

```
SetDialogName("AsdkAcUiSample:AsdkAcUiDialog");
CAcUiDialog::OnInitDialog();
```

Change the constructor's initialization list to also initialize `CAcUiDialog`:

```
AsdkAcUiDialogSample::AsdkAcUiDialogSample
(CWnd* pParent /*=NULL*/)
: CAcUiDialog(AsdkAcUiDialogSample::IDD, pParent)
```

Add message handlers for the `IDC_BUTTON_ANGLE`, `IDC_BUTTON_POINT`, `IDC_COMBO_REGAPPS`, `IDC_EDIT_ANGLE`, and `IDC_EDIT_nPOINT` resources. The handlers should be mapped as follows:

Message handlers

Handler Function	Resource ID	Message
OnButtonAngle	IDC_BUTTON_ANGLE	BN_CLICKED
OnButtonPoint	IDC_BUTTON_POINT	BN_CLICKED
OnKillfocusComboRegapps	IDC_COMBO_REGAPPS	CBN_KILLFOCUS
OnKillfocusEditAngle	IDC_EDIT_ANGLE	EN_KILLFOCUS
OnKillfocusEditXpt	IDC_EDIT_XPOINT	EN_KILLFOCUS
OnKillfocusEditYpt	IDC_EDIT_YPOINT	EN_KILLFOCUS
OnKillfocusEditZpt	IDC_EDIT_ZPOINT	EN_KILLFOCUS

Add Code to the Handlers

Once you have added the handlers, you are ready to add code to deal with your dialog. This section summarizes what each handler does with a complete listing.

To add code to handlers for the MFC dialog

1. Add utility functions to convert, display, and validate the values. Use the `CAcUiNumeric` and `CAcUiAngleEdit` controls to do this:

```
// Utility functions
void AsdkAcUiDialogSample::DisplayPoint()
{
    m_ctrlXPtEdit.SetWindowText(m_strXPt);
    m_ctrlXPtEdit.Convert();
    m_ctrlYPtEdit.SetWindowText(m_strYPt);
    m_ctrlYPtEdit.Convert();
    m_ctrlZPtEdit.SetWindowText(m_strZPt);
    m_ctrlZPtEdit.Convert();
}
bool AsdkAcUiDialogSample::ValidatePoint()
{
    if (!m_ctrlXPtEdit.Validate())
        return false;
    if (!m_ctrlYPtEdit.Validate())
        return false;
    if (!m_ctrlZPtEdit.Validate())
        return false;
    return true;
}
void AsdkAcUiDialogSample::DisplayAngle()
{
    m_ctrlAngleEdit.SetWindowText(m_strAngle);
    m_ctrlAngleEdit.Convert();
}
bool AsdkAcUiDialogSample::ValidateAngle()
{
    if (!m_ctrlAngleEdit.Validate())
        return false;
    return true;
}
```

2. Add utility functions to iterate over two symbol tables and display the names in the two list boxes:

```
void AsdkAcUiDialogSample::DisplayBlocks()
{
    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pBlockTable, AcDb::kForRead);
    // Iterate through the block table and display
    // the names in the list box.
    //
    const char *pName;
    AcDbBlockTableIterator *pBTItr;
    if (pBlockTable->newIterator(pBTItr) == Acad::eOk) {
        while (!pBTItr->done()) {
            AcDbBlockTableRecord *pRecord;
            if (pBTItr->getRecord(pRecord, AcDb::kForRead)
                == Acad::eOk) {
                pRecord->getName(pName);
                m_ctrlBlockListBox.InsertString(-1, pName);
                pRecord->close();
            }
            pBTItr->step();
        }
    }
    pBlockTable->close();
}
void AsdkAcUiDialogSample::DisplayRegApps()
{
    AcDbRegAppTable *pRegAppTable;
    acdbHostApplicationServices()->workingDatabase()
        ->getSymbolTable(pRegAppTable, AcDb::kForRead);
    // Iterate through the reg app table and display the
    // names in the list box.
    //
    const char *pName;
    AcDbRegAppTableIterator *pItr;
    if (pRegAppTable->newIterator(pItr) == Acad::eOk) {
        while (!pItr->done()) {
            AcDbRegAppTableRecord *pRecord;
            if (pItr->getRecord(pRecord, AcDb::kForRead)
                == Acad::eOk) {
                pRecord->getName(pName);
                m_ctrlRegAppComboBox.InsertString(-1, pName);
                pRecord->close();
            }
            pItr->step();
        }
    }
    pRegAppTable->close();
}
```

3. Add the declarations for the functions and variables to the class definition in the header file

```
    void DisplayPoint();
    bool ValidatePoint();
    void DisplayAngle();
    bool ValidateAngle();
    void DisplayBlocks();
    void DisplayRegApps();
    CString m_strAngle;
    CString m_strXPt;
    CString m_strYPt;
    CString m_strZPt;
```

4. Add the button handlers for picking a point and angle using the AutoCAD editor. The `BeginEditorCommand()`, `CompleteEditorCommand()`, and `CancelEditorCommand()` functions are used to hide the dialog, allow the call to `acedGetPoint` and `acedGetAngle`, and finally, either cancel or redisplay the dialog based on how the user picked:

```
// AsdkAcUiDialogSample message handlers
void AsdkAcUiDialogSample::OnButtonPoint()
{
    // Hide the dialog and give control to the editor
    //
    BeginEditorCommand();
    ads_point pt;
    // Get a point
    //
    if (acedGetPoint(NULL, "\nPick a point: ", pt) == RTNORM) {
        // If the point is good, continue
        //
        CompleteEditorCommand();
        m_strXPt.Format("%g", pt[X]);
        m_strYPt.Format("%g", pt[Y]);
        m_strZPt.Format("%g", pt[Z]);
        DisplayPoint();
    } else {
        // otherwise cancel the command (including the dialog)
        CancelEditorCommand();
    }
}
void AsdkAcUiDialogSample::OnButtonAngle()
{
    // Hide the dialog and give control to the editor
    //
    BeginEditorCommand();
    // Set up the default point for picking an angle
    // based on the m_strXPt, m_strYPt, and m_strZPt values
    //
    ads_point pt;
    acdbDisToF(m_strXPt, -1, &pt[X]);
    acdbDisToF(m_strYPt, -1, &pt[Y]);
    acdbDisToF(m_strZPt, -1, &pt[Z]);
    double angle;
    // Get a point from the user
    //
    if (acedGetAngle
        (pt, "\nPick an angle: ", &angle) == RTNORM) {
        // If we got an angle, go back to the dialog
        //
        CompleteEditorCommand();
        // Convert the acquired radian value to degrees since
        // the AcUi control can convert that to the other
        // formats.
        //
        m_strAngle.Format("%g", angle*(180.0/PI));
        DisplayAngle();
    } else {
    // otherwise cancel the command (including the dialog)
    //
        CancelEditorCommand();
    }
}
```

5. Implement the edit box handlers. These functions convert the values to the current Units settings:

```
void AsdkAcUiDialogSample::OnKillfocusEditAngle()
{
    // Get and update text the user typed in.
    //
    m_ctrlAngleEdit.Convert();
    m_ctrlAngleEdit.GetWindowText(m_strAngle);
}
void AsdkAcUiDialogSample::OnKillfocusEditXpt()
{
    // Get and update text the user typed in.
    //
    m_ctrlXPtEdit.Convert();
    m_ctrlXPtEdit.GetWindowText(m_strXPt);
}
void AsdkAcUiDialogSample::OnKillfocusEditYpt()
{
    // Get and update text the user typed in.
    //
    m_ctrlYPtEdit.Convert();
    m_ctrlYPtEdit.GetWindowText(m_strYPt);
}
void AsdkAcUiDialogSample::OnKillfocusEditZpt()
{
    // Get and update text the user typed in.
    //
    m_ctrlZPtEdit.Convert();
    m_ctrlZPtEdit.GetWindowText(m_strZPt);
}
```

6. Implement the combo box handler to allow the user to type in a string, and then register the user's string as an application name. This step, though of little practical value, demonstrates the use of a combo box:

```
void AsdkAcUiDialogSample::OnKillfocusComboRegapps()
{
    CString strFromEdit;
    m_ctrlRegAppComboBox.GetWindowText(strFromEdit);
    if (m_ctrlRegAppComboBox.FindString(-1, strFromEdit) == CB_ERR)
        if (acdbRegApp(strFromEdit) == RTNORM)
            m_ctrlRegAppComboBox.AddString(strFromEdit);
}
```

7. Handle data validation in the `OnOk()` handler for this example. (Of course, it could be done elsewhere.) Also notice that the `OnOk()` handler stores the data in the user profile (registry) using the `SetDialogData()` function:

```
void AsdkAcUiDialogSample::OnOK()
{
    if (!ValidatePoint()) {
        AfxMessageBox("Sorry, Point out of desired range.");
        m_ctrlXPtEdit.SetFocus();
        return;
    }
    if (!ValidateAngle()) {
        AfxMessageBox("Sorry, Angle out of desired range.”);
        m_ctrlAngleEdit.SetFocus();
        return;
    }
    CAcUiDialog::OnOK();
    // Store the data into the registry
    //
    SetDialogData("ANGLE", m_strAngle);
    SetDialogData("POINTX", m_strXPt);
    SetDialogData("POINTY", m_strYPt);
    SetDialogData("POINTZ", m_strZPt);
}
```

8. Finally, the `OnInitDialog()` function takes care of initialization, including the resizing and data persistency requirements:

```
BOOL AsdkAcUiDialogSample::OnInitDialog()
{
    // Set the dialog name for registry lookup and storage
    //
    SetDialogName("AsdkAcUiSample:AsdkAcUiDialog");
    CAcUiDialog::OnInitDialog();
    DLGCTLINFO  dlgSizeInfo[]= {
        { IDC_STATIC_GROUP1, ELASTICX, 20 },
        { IDC_STATIC_GROUP1, ELASTICY, 100 },
        { IDC_EDIT_XPT, ELASTICX, 20 },
        { IDC_EDIT_YPT, ELASTICX, 20 },
        { IDC_EDIT_ZPT, ELASTICX, 20 },
        { IDC_EDIT_ANGLE, ELASTICX, 20 },
        { IDC_STATIC_GROUP2, MOVEX, 20 },
        { IDC_STATIC_GROUP2, ELASTICY, 100 },
        { IDC_STATIC_GROUP2, ELASTICX, 80 },
        { IDC_LIST_BLOCKS, MOVEX, 20 },
        { IDC_LIST_BLOCKS, ELASTICY, 100 },
        { IDC_STATIC_TEXT2, MOVEX, 20 },
        { IDC_STATIC_TEXT2, MOVEY, 100 },
        { IDC_LIST_BLOCKS, ELASTICX, 80 },
        { IDC_STATIC_TEXT2, ELASTICX, 80 },
        { IDC_STATIC_GROUP3, MOVEY, 100 },
        { IDC_STATIC_GROUP3, ELASTICX, 20 },
        { IDC_COMBO_REGAPPS, MOVEY, 100 },
        { IDC_COMBO_REGAPPS, ELASTICX, 20 },
        { IDC_STATIC_TEXT3, MOVEY, 100 },
        { IDC_STATIC_TEXT3, ELASTICX, 20 },
        { IDOK,MOVEX, 100 },
        { IDCANCEL, MOVEX, 100 },
    };
    const DWORD numberofentries =
        sizeof dlgSizeInfo / sizeof DLGCTLINFO;
    SetControlProperty(dlgSizeInfo, numberofentries);
    // Must be within a 100-unit cube centered about 0,0,0.
    //
    m_ctrlXPtEdit.SetRange(-50.0, 50.0);
    m_ctrlYPtEdit.SetRange(-50.0, 50.0);
    m_ctrlZPtEdit.SetRange(-50.0, 50.0);
    // Must be between 0 and 90 degrees.
    //
    m_ctrlAngleEdit.SetRange(0.0, 90.0 /*(PI/2.0)*/);
    // Assign a title for the dialog.
    //
    SetWindowText("AcUiDialog Sample");
    // Load the default bitmaps.
    //
    m_ctrlPickButton.AutoLoad();
    m_ctrlAngleButton.AutoLoad();
    // Get and display the preserved data from the registry.
    //
    if (!GetDialogData("ANGLE", m_strAngle))
        m_strAngle = "0.0";
    if (!GetDialogData("POINTX", m_strXPt))
        m_strXPt = "0.0";
    if (!GetDialogData("POINTY", m_strYPt))
        m_strYPt = "0.0";
    if (!GetDialogData("POINTZ", m_strZPt))
        m_strZPt = "0.0";
    DisplayPoint();
    DisplayAngle();
    DisplayBlocks();
    DisplayRegApps();
    // return TRUE unless you set the focus to a control
    return TRUE;
```

Selection Set, Entity, and Symbol Table Functions

The global functions described in this section handle selection sets, drawing entities, and symbol tables. See the AutoCAD documentation for background information on these topics.

Topics in this section

- Selection Set and Entity Names
- Handling Selection Sets
- Entity Name and Data Functions
- Symbol Table Access

Selection Set and Entity Names

Most of the ObjectARX ® functions that handle selection sets and entities identify a set or entity by its name, which is a pair of longs assigned and maintained by AutoCAD ® . In ObjectARX, names of selection sets and entities have the corresponding type `ads_name`.

Before it can manipulate a selection set or an entity, an ObjectARX application must obtain the current name of the set or entity by calling one of the library functions that returns a selection set or entity name.

Note

Selection set and entity names are volatile; they apply only while you are working on a drawing with AutoCAD, and they are lost when exiting from AutoCAD or switching to another drawing.

For selection sets, which also apply only to the current session, the volatility of names poses no problem, but for entities, which are saved in the drawing database, it does. An application that must refer at different times to the same entities in the same drawing (or drawings), can use entity handles, described in Entity Handles and their Uses.

Handling Selection Sets

The ObjectARX functions that handle selection sets are similar to those in AutoLISP ® . The acedSSGet function provides the most general means of creating a selection set. It creates a selection set in one of three ways:

- Prompting the user to select objects.
- Explicitly specifying the entities to select by using the PICKFIRST set or the Crossing, Crossing Polygon, Fence, Last, Previous, Window, or Window Polygon options (as in interactive AutoCAD use), or by specifying a single point or a fence of points.

- Filtering the current drawing database by specifying a list of attributes and conditions that the selected entities must match. You can use filters with any of the previous options.

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

The first argument to `acedSSGet()` is a string that describes which selection options to use, as summarized in the following table.

Selection options for acedSSGet

Selection Code	Description
NULL	Single-point selection (if pt1 is specified) or user selection (if pt1 is also NULL)
#	Nongeometric (all, last, previous)
:$	Prompts supplied
.	User pick
:?	Other callbacks
A	All
B	Box
C	Crossing
CP	Crossing Polygon
:D	Duplicates OK
:E	Everything in aperture
F	Fence
G	Groups
I	Implied
:K	Keyword callbacks
L	Last
M	Multiple
P	Previous

:S	Force single object selection only
W	Window
WP	Window Polygon
X	Extended search (search whole database)

The next two arguments specify point values for the relevant options. (They should be NULL if they don't apply.) If the fourth argument, entmask, is not NULL, it points to the list of entity field values used in filtering. The fifth argument, ss, identifies the selection set's name.

The following code shows representative calls to acedSSGet(). As the acutBuildList() call illustrates, for the polygon options "CP" and "WP" (but not for "F"), acedSSGet() automatically closes the list of points. You don't need to build a list that specifies a final point identical to the first.

```
ads_point pt1, pt2, pt3, pt4;
struct resbuf *pointlist;
ads_name ssname;
pt1[X] = pt1[Y] = pt1[Z] = 0.0;
pt2[X] = pt2[Y] = 5.0; pt2[Z] = 0.0;
// Get the current PICKFIRST set, if there is one;
// otherwise, ask the user for a general entity selection.
acedSSGet(NULL, NULL, NULL, NULL, ssname);
// Get the current PICKFIRST set, if there is one.
acedSSGet("I", NULL, NULL, NULL, ssname);
// Selects the most recently selected objects.
acedSSGet("P", NULL, NULL, NULL, ssname);
// Selects the last entity added to the database.
acedSSGet("L", NULL, NULL, NULL, ssname);
// Selects entity passing through point (5,5).
acedSSGet(NULL, pt2, NULL, NULL, ssname);
// Selects entities inside the window from (0,0) to (5,5).
acedSSGet("W", pt1, pt2, NULL, ssname);
// Selects entities enclosed by the specified polygon.
pt3[X] = 10.0; pt3[Y] = 5.0; pt3[Z] = 0.0;
pt4[X] = 5.0; pt4[Y] = pt4[Z] = 0.0;
pointlist = acutBuildList(RTPOINT, pt1, RTPOINT, pt2,
        RTPOINT, pt3, RTPOINT, pt4, 0);
acedSSGet("WP", pointlist, NULL, NULL, ssname);
// Selects entities crossing the box from (0,0) to (5,5).
acedSSGet("C", pt1, pt2, NULL, ssname);
// Selects entities crossing the specified polygon.
acedSSGet("CP", pointlist, NULL, NULL, ssname);
acutRelRb(pointlist);
// Selects the entities crossed by the specified fence.
pt4[Y] = 15.0; pt4[Z] = 0.0;
pointlist = acutBuildList(RTPOINT, pt1, RTPOINT, pt2,
        RTPOINT, pt3, RTPOINT, pt4, 0);
acedSSGet("F", pointlist, NULL, NULL, ssname);
acutRelRb(pointlist);
```

The complement of acedSSGet() is acedSSFree, which releases a selection set once the application has finished using it. The selection set is specified by name. The following code fragment uses the ads_name declaration from the previous example.

```
acedSSFree(ssname);
```

Note AutoCAD cannot have more than 128 selection sets open at once. This limit includes the selection sets open in all concurrently running ObjectARX and AutoLISP applications. The limit may be different on your system. If the limit is reached, AutoCAD refuses to create more selection sets. Simultaneously managing a large number of selection sets is not recommended. Instead, keep a reasonable number of sets open at any given time, and call `acedSSFree()` to free unused selection sets as soon as possible. Unlike AutoLISP, the ObjectARX environment has no automatic garbage collection to free selection sets after they have been used. An application should always free its open selection sets when it receives a `kUnloadDwgMsg`, `kEndMsg`, or `kQuitMsg` message.

Topics in this section

- Selection Set Filter Lists
- Selection Set Manipulation
- Transformation of Selection Sets

Selection Set Filter Lists

When the `entmask` argument specifies a list of entity field values, `acedSSGet()` scans the selected entities and creates a selection set containing the names of all main entities that match the specified criteria. For example, using this mechanism, you can obtain a selection set that includes all entities of a given type, on a given layer, or of a given color.

You can use a filter in conjunction with any of the selection options. The "`X`" option says to create the selection set using only filtering; as in previous AutoCAD versions, if you use the "`X`" option, `acedSSGet()` scans the entire drawing database.

Note If only filtering is specified ("`X`") but the `entmask` argument is `NULL`, `acedSSGet()` selects all entities in the database.

The `entmask` argument must be a result buffer list. Each buffer specifies a property to check and a value that constitutes a match; the buffer's `restype` field is a DXF group code that indicates the kind of property to look for, and its `resval` field specifies the value to match.

The following are some examples.

```
struct resbuf eb1, eb2, eb3;
char sbuf1[10], sbuf2[10]; // Buffers to hold strings
ads_name ssname1, ssname2;
eb1.restype = 0;// Entity name
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1;
eb1.rbnext = NULL; // No other properties
// Retrieve all circles.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
eb2.restype = 8; // Layer name
strcpy(sbuf2, "FLOOR3");
eb2.resval.rstring = sbuf2;
eb2.rbnext = NULL; // No other properties
// Retrieve all entities on layer FLOOR3.
acedSSGet("X", NULL, NULL, &eb2, ssname2);
```

Note The `resval` specified in each buffer must be of the appropriate type. For example, name types are strings (`resval.rstring`); elevation and thickness are double-precision floating-point values (`resval.rreal`); color, attributes-follow, and flag values are short integers (`resval.rint`); extrusion vectors are three-dimensional points (`resval.rpoint`); and so forth.

If `entmask` specifies more than one property, an entity is included in the selection set only if it matches *all* specified conditions, as shown in the following example:

```
eb3.restype = 62; // Entity color
eb3.resval.rint = 1; // Request red entities.
eb3.rbnext = NULL; // Last property in list
eb1.rbnext = &eb2; // Add the two properties
eb2.rbnext = &eb3; // to form a list.
// Retrieve all red circles on layer FLOOR3.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

An entity is tested against all fields specified in the filtering list unless the list contains relational or conditional operators, as described inRelational Testsand Conditional Filtering.

The `acedSSGet()` function returns `RTERROR` if no entities in the database match the specified filtering criteria.

The previous `acedSSGet()` examples use the "`X`" option, which scans the entire drawing database. If filter lists are used in conjunction with the other options (user selection, a window, and so forth), the filter is applied only to the entities initially selected.

The following is an example of the filtering of user-selected entities.

```
eb1.restype = 0; // Entity type group
strcpy(sbuf1, "TEXT");
eb1.resval.rstring = sbuf1; // Entity type is text.
eb1.rbnext = NULL;
// Ask the user to generally select entities, but include
// only text entities in the selection set returned.
acedSSGet(NULL, NULL, NULL, &eb1, ssname1);
```

The next example demonstrates the filtering of the previous selection set.

```
eb1.restype = 0; // Entity type group
strcpy(sbuf1, "LINE");
eb1.resval.rstring = sbuf1; // Entity type is line.
eb1.rbnext = NULL;
// Select all the lines in the previously created selection set.
acedSSGet("P", NULL, NULL, &eb1, ssname1);
```

The final example shows the filtering of entities within a selection window.

```
eb1.restype = 8; // Layer
strcpy(sbuf1, "FLOOR9");
eb1.resval.rstring = sbuf1; // Layer name
eb1.rbnext = NULL;
// Select all the entities within the window that are also
// on the layer FLOOR9.
acedSSGet("W", pt1, pt2, &eb1, ssname1);
```

Note The meaning of certain group codes can differ from entity to entity, and not all group codes are present in all entities. If a particular group code is specified in a filter, entities that do not contain that group code are excluded from the selection sets that `acedSSGet()` returns.

Topics in this section

- Wild-Card Patterns in Filter Lists
- Filtering for Extended Data
- Relational Tests
- Conditional Filtering

Wild-Card Patterns in Filter Lists

Symbol names specified in filter lists can include wild-card patterns. The wild-card patterns recognized by `acedSSGet()` are the same as those recognized by the function `acutWcMatch()`.

The following sample code retrieves an anonymous block named `*U2`.

```
eb2.restype = 2; // Block name
strcpy(sbuf1, "'*U2"); // Note the reverse quote.
eb2.resval.rstring = sbuf1; // Anonymous block name
eb2.rbnext = NULL;
// Select Block Inserts of the anonymous block *U2.
acedSSGet("X", NULL, NULL, &eb2, ssname1);
```

Filtering for Extended Data

Extended data (xdata) are text strings, numeric values, 3D points, distances, layer names, or other data attached to an object, typically by an external application.

The size of extended data is 16K bytes.

You can retrieve extended data for a particular application by specifying its name in a filter list, using the -3 group code. The acedSSGet() function returns entities with extended data registered to the specified name; acedSSGet() does not retrieve individual extended data items (with group codes in the range 1000–2000).

The following sample code fragment selects all circles that have extended data registered to the application whose ID is "APPNAME".

```
eb1.restype = 0; // Entity type
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1; // Circle
eb1.rbnext = &eb2;
eb2.restype = -3; // Extended data
eb2.rbnext = &eb3;
eb3.restype = 1001;
strcpy(sbuf2, "APPNAME");
eb3.resval.rstring = sbuf2; // APPNAME application
eb3.rbnext = NULL;
// Select circles with XDATA registered to APPNAME.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

If more than one application name is included in the list, acedSSGet() includes an entity in the selection set only if it has extended data for all the specified applications. For example, the following code selects circles with extended data registered to "APP1" and "APP2".

```
eb1.restype = 0; // Entity type
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1; // Circle
eb1.rbnext = &eb2;
eb2.restype = -3; // Extended data
eb2.rbnext = &eb3;
eb3.restype = 1001;
strcpy(sbuf2, "APP1");
eb2.resval.rstring = sbuf2; // APP1 application
eb2.rbnext = &eb4;
eb4.restype = 1001; // Extended data
strcpy(sbuf3, "APP2");
eb4.resval.rstring = sbuf3; // APP2 application
eb4.rbnext = NULL;
// Select circles with XDATA registered to APP1 & APP2.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

You can specify application names using wild-card strings, so you can search for the data of multiple applications at one time. For example, the following code selects all circles with extended data registered to "APP1" or "APP2" (or both).

```
eb1.restype = 0; // Entity type
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1; // Circle
eb1.rbnext = &eb2;
eb2.restype = -3; // Extended data
eb2.rbnext = &eb3;
eb3.restype = 1001; // Extended data
strcpy(sbuf2, "APP1,APP2");
eb3.resval.rstring = sbuf2; // Application names
eb3.rbnext = NULL;
// Select circles with XDATA registered to APP1 or APP2.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

The following string finds extended data of the same application.

```
strcpy(sbuf2, "APP[12]");
```

Relational Tests

Unless you specify otherwise, there is an implied "equals" test between the entity and each item in the filter list. For numeric groups (integers, real values, points, and vectors), you can specify other relations by including relational operators in the filter list. Relational operators are passed as a special –4 group, whose value is a string that indicates the test to be applied to the next group in the filter list.

The following sample code selects all circles whose radii are greater than or equal to 2.0:

```
eb3.restype = 40; // Radius
eb3.resval.rreal = 2.0;
eb3.rbnext = NULL;
eb2.restype = -4; // Filter operator
strcpy(sbuf1, ">=");
eb2.resval.rstring = sbuf1; // Greater than or equals
eb2.rbnext = &eb3;
eb1.restype = 0; // Entity type
strcpy(sbuf2, "CIRCLE");
eb1.resval.rstring = sbuf2; // Circle
eb1.rbnext = &eb2;
// Select circles whose radius is >= 2.0.
acedSSGet("X", NULL, NULL, &eb1, ssname1);
```

Conditional Filtering

The relational operators just described are binary operators. You can also test groups by creating nested Boolean expressions that use conditional operators. The conditional operators are also specified by -4 groups, but they must be paired.

The following sample code selects all circles in the drawing with a radius of 1.0 and all lines on the layer "ABC".

```
struct resbuf* prb;
prb = acutBuildList(-4, "<or",-4, "<and", RTDXF0,
    "CIRCLE", 40, 1.0, -4, "and>", -4, "<and", RTDXF0,
    "LINE", 8, "ABC", -4, "and>", -4, "or>", 0);
acedSSGet("X", NULL, NULL, prb, ssname1);
```

The conditional operators are not case sensitive; you can use lowercase equivalents.

Note Conditional expressions that test for extended data using the -3 group can contain only -3 groups. See Filtering for Extended Data.

To select all circles that have extended data registered to either "APP1" or "APP2" but not both, you could use the following code.

```
prb = acutBuildList(-4, "<xor", -3, "APP1", -3, "APP2",
    -4, "xor>", 0);
acedSSGet("X", NULL, NULL, prb, ssname1);
```

Selection Set Manipulation

You can add entities to a selection set or remove them from it by calling the functions `acedSSAdd()` and `acedSSDel()`, which are similar to the Add and Remove options when AutoCAD interactively prompts the user to select objects or remove objects.

Note The `acedSSAdd()` function can also be used to create a new selection set, as shown in the following example. As with `acedSSGet()`, `acedSSAdd()` creates a new selection set only if it returns `RTNORM`.

The following sample code fragment creates a selection set that includes the first and last entities in the current drawing.

```
ads_name fname, lname; // Entity names
ads_name ourset; // Selection set name
// Get the first entity in the drawing.
if (acdbEntNext(NULL, fname) != RTNORM) {
    acdbFail("No entities in drawing\n");
    return BAD;
}
// Create a selection set that contains the first entity.
if (acedSSAdd(fname, NULL, ourset) != RTNORM) {
    acdbFail("Unable to create selection set\n");
    return BAD;
}
// Get the last entity in the drawing.
if (acdbEntLast(lname) != RTNORM) {
    acdbFail("No entities in drawing\n");
    return BAD;
}
// Add the last entity to the same selection set.
if (acedSSAdd(lname, ourset, ourset) != RTNORM) {
    acdbFail("Unable to add entity to selection set\n");
    return BAD;

}
```

The example runs correctly even if there is only one entity in the database (in which case both `acdbEntNext()` and `acdbEntLast()` set their arguments to the same entity name). If `acedSSAdd()` is passed the name of an entity that is already in the selection set, it ignores the request and does not report an error.

As the example also illustrates, the second and third arguments to `acedSSAdd()` can be passed as the same selection set name. That is, if the call is successful, the selection set named by both arguments contains an additional member after `acedSSAdd()` returns (unless the specified entity was already in the selection set).

The following call removes the entity with which the selection set was created in the previous example.

```
acedSSDel(fname, ourset);
```

If there is more than one entity in the drawing (that is, if `fname` and `lname` are not equal), the selection set `ourset` now contains only `lname`, the last entity in the drawing.

The function `acedSSLength()` returns the number of entities in a selection set, and `acedSSMemb()` tests whether a particular entity is a member of a selection set. Finally, the function `acedSSName()` returns the name of a particular entity in a selection set, using an index into the set (entities in a selection set are numbered from 0).

Note Because selection sets can be quite large, the `len` argument returned by `acedSSLength()` must be declared as a `long` integer. The `i` argument used as an index in calls to `acedSSName()` must also be a `long` integer. (In this context, standard C compilers will correctly convert a plain integer.)

The following sample code shows a few calls to `acedSSName()`.

```
ads_name sset, ent1, ent4, lastent;
long ilast;
// Create the selection set (by prompting the user).
acedSSGet(NULL, NULL, NULL, NULL, sset);
// Get the name of first entity in sset.
if (acedSSName(sset, 0L, ent1) != RTNORM)
    return BAD;
// Get the name of the fourth entity in sset.
if (acedSSName(sset, 3L, ent4) != RTNORM) {
    acdbFail("Need to select at least four entities\n");
    return BAD;
}
// Find the index of the last entity in sset.
if (acedSSLength(sset, &ilast) != RTNORM)
    return BAD;
// Get the name of the last entity in sset.
if (acedSSName(sset, ilast-1, lastent) != RTNORM)
    return BAD;
```

Transformation of Selection Sets

The function `acedXformSS()` transforms a selection set by applying a transformation matrix (of type `ads_matrix`) to the entities in the set. This provides an efficient alternative to invoking the ROTATE, SCALE, MIRROR, or MOVE commands with `acedCommand()` (or `acedCmd()`) or to changing values in the database with `acdbEntMod()`. The selection set can be obtained in any of the usual ways. The matrix must do uniform scaling. That is, the elements in the scaling vector $S_X\ S_Y\ S_Z$ must all be equal; in matrix notation, $M_{00}\ M_{11}\ M_{22}$.

If the scale vector is not uniform, `acedXformSS()` reports an error.

The following sample code gets a selection set by using a crossing box, and then applies the following matrix to it.

$$\begin{bmatrix} 0.5 & 0.0 & 0.0 & 20.0 \\ 0.0 & 0.5 & 0.0 & 5.0 \\ 0.0 & 0.0 & 0.5 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Applying this matrix scales the entities by one-half (which moves them toward the origin) and translates their location by (20.0,5.0).

```
int rc, i, j;
ads_point pt1, pt2;
ads_matrix matrix;
ads_name ssname;
// Initialize pt1 and pt2 here.
rc = acedSSGet("C", pt1, pt2, NULL, ssname);
if (rc == RTNORM) {
// Initialize to identity.
    ident_init(matrix);
// Initialize scale factors.
    matrix[0][0] = matrix[1][1] = matrix[2][2] = 0.5;
// Initialize translation vector.
    matrix[0][T] = 20.0;
    matrix[1][T] = 5.0;
    rc = acedXformSS(ssname, matrix);
}
```

When you invoke acedDragGen(), you must specify a similar function to let users interactively control the effect of the transformation. The function's declaration must have the following form:

```
int scnf(ads_point pt, ads_matrix mt)
```

It should return RTNORM if it modified the matrix, RTNONE if it did not, or RTERROR if it detects an error.

The acedDragGen() function calls the scnf function every time the user moves the cursor. The scnf() function sets the new value of the matrix mt. When scnf() returns with a status of RTNORM, acedDragGen() applies the new matrix to the selection set. If there is no need to modify the matrix (for example, if scnf() simply displays transient vectors with acedGrVecs()), scnf() should return RTNONE. In this case, acedDragGen() ignores mt and doesn't transform the selection set.

In the following example, the function sets the matrix to simply move (translate) the selection set without scaling or rotation.

```
int dragsample(usrpt, matrix)
ads_point usrpt
ads_matrix matrix;
{
    ident_init(matrix); // Initialize to identity.
// Initialize translation vector.
    matrix[0][T] = usrpt[X];
    matrix[1][T] = usrpt[Y];
    matrix[2][T] = usrpt[Z];
    return RTNORM; // Matrix was modified.
}
```

Conversely, the following version of dragsample() scales the selection set in the current *XY* plane but doesn't move it.

```
int dragsample(usrpt, matrix)
ads_point usrpt
ads_matrix matrix;
{
    ident_init(matrix); // Initialize to identity.
    matrix[0][0] = userpt[X];
    matrix[1][1] = userpt[Y];
    return RTNORM; // Matrix was modified.
}
```

A call to `acedDragGen()` that employs the transformation function looks like this:

```
int rc;
ads_name ssname;
ads_point return_pt;
// Prompt the user for a general entity selection:
if (acedSSGet(NULL, NULL, NULL, NULL, ssname) == RTNORM)
    rc = acedDragGen(ssname, // The new entities
        "Scale the selected objects by dragging", // Prompt
         0, // Display normal cursor (crosshairs)
         dragsample, // Pointer to the transform function
         return_pt); // Set to the specified location
```

More complex transformations can rotate entities, combine transformations (as in the `acedXformSS()` example), and so forth.

Combining transformation matrices is known as matrix composition. The following function composes two transformation matrices by returning their product in `resmat`.

```
void xformcompose(ads_matrix xf1, ads_matrix xf2,
        ads_matrix resmat)
{
    int i, j, k;
    ads_real sum;
    for (i=0; i<=3; i++) {
        for (j=0; j<=3; j++) {
            sum = 0.0;
            for (k=0; k<3; k++) {
                sum += xf1[i,k] * xf2[k,j];
            }
            resmat[i,j] = sum;
        }
    }
}
```

Entity Name and Data Functions

Entity-handling functions are organized into two categories: functions that retrieve the name of a particular entity and functions that retrieve or modify entity data.

Topics in this section

- Entity Name Functions
- Entity Data Functions
- Entity Data Functions and Graphics Screen
- Notes on Extended Data
- Xrecord Objects

Entity Name Functions

To operate on an entity, an ObjectARX application must obtain its name for use in subsequent calls to the entity data functions or the selection set functions. The functions `acedEntSel()`, `acedNEntSelP()`, and `acedNEntSel()` return not only the entity's name but additional information for the application's use. The `entsel` functions require AutoCAD users (or the application) to select an entity by specifying a point on the graphics screen; all the other entity name functions can retrieve an entity even if it is not visible on the screen or is on a frozen layer. Like the `acedGetxxx()` functions, you can have `acedEntSel()`, `acedNEntSelP()`, and `acedNEntSel()` return a keyword instead of a point by preceding them with a call to `acedInitGet()`.

If a call to `acedEntSel()`, `acedNEntSelP()`, or `acedNEntSel()` returns RTERROR, and you want to know whether the user specified a point that had no entity or whether the user pressed RETURN, you can inspect the value of the ERRNO system variable. If the user specified an empty point, ERRNO equals 7 (OL_ENTSELPICK). If the user pressed RETURN, ERRNO equals 52 (OL_ENTSELNULL). (You can use the symbolic names if your program includes the header file ol_*errno.h*.)

Note You should inspect ERRNO immediately after `acedEntSel()`, `acedNEntSelP()`, or `acedNEntSel()` returns. A subsequent ObjectARX call can change the value of ERRNO.

The `acdbEntNext()` function retrieves entity names sequentially. If its first argument is `NULL`, it returns the name of the first entity in the drawing database; if its first argument is the name of an entity in the current drawing, it returns the name of the succeeding entity.

The following sample code fragment illustrates how `acedSSAdd()` can be used in conjunction with `acdbEntNext()` to create selection sets and to add members to an existing set.

```
ads_name ss, e1, e2;
// Set e1 to the name of first entity.
if (acdbEntNext(NULL, e1) != RTNORM) {
    acdbFail("No entities in drawing\n");
    return BAD;
}
// Set ss to a null selection set.
acedSSAdd(NULL, NULL, ss);
// Return the selection set ss with entity name e1 added.
if (acedSSAdd(e1, ss, ss) != RTNORM) {
    acdbFail("Unable to add entity to selection set\n");
    return BAD;
}
// Get the entity following e1.
if (acdbEntNext(e1, e2) != RTNORM) {
    acdbFail("Not enough entities in drawing\n");
    return BAD;
}
// Add e2 to selection set ss
if (acedSSAdd(e2, ss, ss) != RTNORM) {
    acdbFail("Unable to add entity to selection set\n");
    return BAD;
}
```

The following sample code fragment uses `acdbEntNext()` to "walk" through the database, one entity at a time.

```
ads_name ent0, ent1;
struct resbuf *entdata;
if (acdbEntNext(NULL, ent0) != RTNORM) {
    acdbFail("Drawing is empty\n");
    return BAD;
}
do {
// Get entity's definition data.
    entdata = acdbEntGet(ent0);
    if (entdata == NULL) {
        acdbFail("Failed to get entity\n");
        return BAD;
    }
        .
        . // Process new entity.
        .
    if (acedUsrBrk() == TRUE) {
        acdbFail("User break\n");
        return BAD;
    }
    acutRelRb(entdata); // Release the list.
    ads_name_set(ent0, ent1); // Bump the name.
} while (acdbEntNext(ent1, ent0) == RTNORM);
```

Note You can also go through the database by "bumping" a single variable in the `acdbEntNext()` call (such as `acdbEntNext(ent0, ent0)`), but if you do, the value of the variable is no longer defined once the loop ends.

The `acdbEntLast()` function retrieves the name of the last entity in the database. The last entity is the most recently created main entity, so `acdbEntLast()` can be called to obtain the name of an entity that has just been created by means of a call to `acedCommand()`, `acedCmd()`, or `acdbEntMake()`.

The acedEntSel() function prompts the AutoCAD user to select an entity by specifying a point on the graphics screen; acedEntSel() returns both the entity name and the value of the specified point. Some entity operations require knowledge of the point by which the entity was selected. Examples from the set of existing AutoCAD commands include BREAK, TRIM, EXTEND, and OSNAP.

Topics in this section

- Entity Handles and their Uses
- Entity Context and Coordinate Transform Data

Entity Handles and their Uses

The acdbHandEnt() function retrieves the name of an entity with a specific handle. Like entity names, handles are unique within a drawing. Unlike entity names, an entity's handle is constant throughout its life. ObjectARX applications that manipulate a specific database can use acdbHandEnt() to obtain the current name of an entity they must use.

The following sample code fragment uses acdbHandEnt() to obtain an entity name and to print it out.

```
char handle[17];
ads_name e1;
strcpy(handle, "5a2");
if (acdbHandEnt(handle, e1) != RTNORM)
    acdbFail("No entity with that handle exists\n");
else
    acutPrintf("%ld", e1[0]);
```

In one particular editing session, this code might print out 60004722. In another editing session with the same drawing, it might print an entirely different number. But in both cases, the code is accessing the same entity.

The acdbHandEnt() function has an additional use: entities deleted from the database (with acdbEntDel()) are not purged until you leave the current drawing (by exiting AutoCAD or switching to another drawing). This means that acdbHandEnt() can recover the names of deleted entities, which can then be restored to the drawing by a second call to acdbEntDel().

Entities in drawings cross-referenced with XREF Attach are not actually part of the current drawing; their handles are unchanged and cannot be accessed by acdbHandEnt(). However, when drawings are combined by means of INSERT, INSERT *, XREF Bind (XBIND), or partial DXFIN, the handles of entities in the incoming drawing are lost, and incoming entities are assigned new handle values to ensure that each handle in the original drawing remains unique.

Note Extended data can include entity handles to save relational structures in a drawing. In some circumstances, these handles require translation or maintenance. See Using Handles in Extended Data.

Entity Context and Coordinate Transform Data

The acedNEntSelP() function is similar to acedEntSel(), except that it passes two additional result arguments to facilitate the handling of entities that are nested within block references.

Another difference between acedNEntSelP() and acedEntSel() is that when the user responds to an acedNEntSelP() call by selecting a complex entity, acedNEntSelP() returns the selected subentity and not the complex entity's header as acedEntSel() does. For example, when the user selects a polyline, acedNEntSelP() returns a vertex subentity instead of the polyline header. To retrieve the polyline header, the application must use acdbEntNext() to walk forward to the Seqend subentity and obtain the name of the header from the Seqend subentity's -2 group. This is true also when the user selects attributes in a nested block reference and when the pick point is specified in the acedNEntSelP() call.

Topics in this section

- Coordinate Transformation
- Context Data

Coordinate Transformation

The first of the additional arguments returned by acedNEntSelP() is a 4x4 transformation matrix of type ads_matrix. This matrix is known as the Model to World Transformation Matrix. It enables the application to transform points in the entity's definition data (and extended data, if that is present) from the entity's model coordinate system (MCS) into the World Coordinate System (WCS). The MCS applies only to nested entities. The origin of the MCS is the insert point of the block, and its orientation is that of the UCS that was in effect when the block was created.

If the selected entity is not a nested entity, the transformation matrix is set to the identity matrix. The transformation is expressed by the following matrix multiplication:

$$X' = M_{00}X + M_{01}Y + M_{02}Z + M_{03}$$

$$Y' = M_{10}X + M_{11}Y + M_{12}Z + M_{13}$$

$$Z' = M_{20}X + M_{21}Y + M_{22}Z + M_{23}$$

The individual coordinates of a transformed point are obtained from the equations where M_{mn} is the Model to World Transformation Matrix coordinates, (X,Y,Z) is the entity definition data point expressed in MCS coordinates, and (X',Y',Z') is the resulting entity definition data point expressed in WCS coordinates. See Transformation Matrices on page 551.

Note To transform a vector rather than a point, do not add the translation vector [M_{03} M_{13} M_{23}] (from the fourth column of the transformation matrix).

The following sample code defines a function, mcs2wcs(), that performs the transformations described by the preceding equations. It takes the transformation matrix returned by acedNEntSelP() and a single point (presumably from the definition data of a nested entity), and returns the translated point. If the third argument to mcs2wcs(), is_pt, is set to 0 (FALSE), the last column of the transformation matrix—the translation

vector or displacement—is not added to the result. This enables the function to translate a vector as well as a point.

```
void mcs2wcs(xform, entpt, is_pt, worldpt)
ads_matrix xform;
ads_point entpt, worldpt;
int is_pt;
{
    int i, j;
    worldpt[X] = worldpt[Y] = worldpt[Z] = 0.0;
    for (i=X; i<=Z; i++)
        for (j=X; j<=Z; j++)
            worldpt[i] += xform[i][j] * entpt[j];
    if (is_pt) // If it's a point, add in the displacement
        for (i=X; i<=Z; i++)
            worldpt[i] += xform[i][T];
}
```

The following code fragment shows how `mcs2wcs()` might be used in conjunction with `acedNEntSelP()` to translate point values into the current WCS.

```
ads_name usrent, containent;
ads_point usrpt, defpt, wcspt;
ads_matrix matrix;
struct resbuf *containers, *data, *rb, *prevrb;
status = acedNEntSelP(NULL, usrent, usrpt, FALSE, matrix,
    &containers);
if ((status != RTNORM) || (containers == NULL))
    return BAD;
data = acdbEntGet(usrent);
// Extract a point (defpt) from the data obtained by calling
// acdbEntGet() for the selected kind of entity.
.
.
.
mcs2wcs(matrix, defpt, TRUE, wcspt);
```

The `acedNEntSelP()` function also allows the program to specify the pick point. A `pickflag` argument determines whether or not `acedNEntSelP()` is called interactively.

In the following example, the `acedNEntSelP()` call specifies its own point for picking the entity and does not prompt the user. The `pickflag` argument is `TRUE` to indicate that the call supplies its own point value (also, the `prompt` is `NULL`).

```
ads_point ownpoint;
ownpoint[X] = 2.7; ownpoint[Y] = 1.5; ownpoint[Z] = 0.0;
status = acedNEntSelP(NULL, usrent, ownpt, TRUE, matrix,
    &containers);
```

The `acedNEntSel()` function is provided for compatibility with existing ObjectARX applications. New applications should be written using `acedNEntSelP()`.

The Model to World Transformation Matrix returned by the call to `acedNEntSel()` has the same purpose as that returned by `acedNEntSelP()`, but it is a 4x3 matrix—passed as an array of four points—that uses the convention that a point is a row rather than a column. The transformation is described by the following matrix multiplication:

$$\begin{bmatrix} X' & Y' & Z' & 1.0 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1.0 \end{bmatrix} \bullet \begin{bmatrix} M_{00} & M_{10} & M_{20} \\ M_{01} & M_{11} & M_{21} \\ M_{02} & M_{12} & M_{22} \\ M_{03} & M_{13} & M_{23} \end{bmatrix}$$

The equations for deriving the new coordinates are as follows:

$$X' = XM_{00} + YM_{01} + ZM_{02} + M_{03}$$

$$Y' = XM_{10} + YM_{11} + ZM_{12} + M_{13}$$

$$Z' = XM_{20} + YM_{21} + ZM_{22} + M_{23}$$

Although the matrix format is different, the formulas are equivalent to those for the `ads_matrix` type, and the only change required to adapt `mcs2wcs()` for use with `acedNEntSel()` is to declare the matrix argument as an array of four points.

```
void mcs2wcs(xform, entpt, is_pt, worldpt);
ads_point xform[4]; // 4x3 version
ads_point entpt, worldpt;
int is_pt;
```

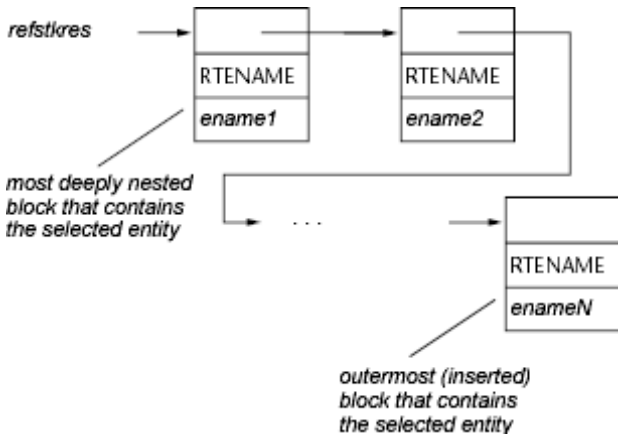
The identity form of the 4x3 matrix is as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

In addition to using a different matrix convention, `acedNEntSel()` doesn't let the program specify the pick point.

Context Data

The function `acedNEntSelP()` provides an argument for context data, `refstkres`. (This is another feature not provided by `acedEntSel()`.) The `refstkres` argument is a pointer to a linked list of result buffers that contains the names of the entity's container blocks. The list is ordered from lowest to highest. In other words, the first name in the list is the name of the block containing the selected entity, and the last name in the list is the name of the block that was directly inserted into the drawing. The following figure shows the format of this list.



If the selected entity `entres` is not a nested entity, `refstkres` is a `NULL` pointer. This is a convenient way to test whether or not the entity's coordinates need to be translated. (Because `xformres` is returned as the identity matrix for entities that are not nested, applying it to the coordinates of such entities does no harm but does cost some needless execution time.)

Using declarations from the previous `acedNEntSelP()` example, the name of the block that immediately contains the user-selected entity can be found by the following code (in the `acedNEntSelP()` call, the `pickflag` argument is `FALSE` for interactive selection).

```
status = acedNEntSelP(NULL, usrent, usrpt, FALSE, matrix,
    &containers);
if ((status != RTNORM) || (containers == NULL))
    return BAD;
containent[0] = containers->resval.rlname[0];
containent[1] = containers->resval.rlname[1];
```

The name of the outermost container (that is, the entity originally inserted into the drawing) can be found by a sequence such as the following:

```
// Check that containers is not already NULL.
rb = containers;
while (rb != NULL) {
    prevrb = rb;
    rb = containers->rbnext;
}
// The result buffer pointed to by prevrb now contains the
// name of the outermost block.
```

In the following example, the current coordinate system is the WCS. Using AutoCAD, create a block named SQUARE consisting of four lines.

Command: **line**

Specify first point: **1,1**

Specify next point or [Undo]: **3,1**

Specify next point or [Undo]: **3,3**

Specify next point or [Close/Undo]: **1,3**

Specify next point or [Close/Undo]: **c**

Command: -**block**

Enter block name or [?]: **square**

Specify insertion base point: **2,2**

Select objects: *Select the four lines you just drew*

Select objects: ENTER

Then insert the block in a UCS rotated 45 degrees about the *Z* axis.

Command: **ucs**

Enter an option [New/Move/orthoGraphic/Prev/Restore/Save/Del/Apply/?/World] <World>: **z**

Specify rotation angle about Z axis <90>: **45**

Command: -**insert**

Enter block name or [?]: **square**

Specify insertion point or [Scale/X/Y/Z/Rotate/PScale/PX/PY/PZ/PRotate]:**7,0**

Enter X scale factor, specify opposite corner, or [Corner/XYZ] <1>: ENTER

Enter Y scale factor <use X scale factor>: ENTER

Specify rotation angle <0>: ENTER

If an ObjectARX application calls `acedNEntSelP()` (or `acedNEntSel()`) and the user selects the lower-left side of the square, these functions set the `entres` argument to equal the name of the selected line. They set the pick point (`ptres`) to (6.46616,-1.0606,0.0) or a nearby point value. They return the transformation matrix (`xformres`) as shown in the following figure. Finally, they set the list of container entities (`refstkres`) to point to a single result buffer containing the entity name of the block SQUARE.

0.707107	-0.707107	0.0	4.94975
0.707107	0.707107	-0.0	4.94975
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

ads_nentselp() result

0.707107	0.707107	0.0
-0.707107	0.707107	0.0
0.0	-0.0	1.0
4.94975	4.94975	0.0

ads_nentsel() result

Entity Data Functions

Some functions operate on entity data and can be used to modify the current drawing database. The `acdbEntDel()` function deletes a specified entity. The entity is not purged from the database until you leave the current drawing. So if the application calls `acdbEntDel()` a second time during that session and specifies the same entity, the entity is undeleted. (You can use `acdbHandEnt()` to retrieve the names of deleted entities.)

Note Using `acdbEntDel()`, attributes and polyline vertices cannot be deleted independently from their parent entities; `acdbEntDel()` operates only on main entities. To delete an attribute or vertex, use `acedCommand()` or `acedCmd()` to invoke the AutoCAD ATTEDIT or PEDIT commands, use `acdbEntMod()` to redefine the entity without the unwanted subentities, or open the vertex or attribute and use its `erase()` method to erase it.

The `acdbEntGet()` function returns the definition data of a specified entity. The data is returned as a linked list of result buffers. The type of each item (buffer) in the list is specified by a DXF group code. The first item in the list contains the entity's current name (`restype == -1`).

An ObjectARX application could retrieve and print the definition data for an entity by using the following two functions. (The `printdxf()` function does not handle extended data.)

```
void getlast()
{
    struct resbuf *ebuf, *eb;
    ads_name ent1;
    acdbEntLast(ent1);
    ebuf = acdbEntGet(ent1);
    eb = ebuf;
    acutPrintf("\nResults of entgetting last entity\n");
    // Print items in the list.
    for (eb = ebuf; eb != NULL; eb = eb->rbnext)
        printdxf(eb);
    // Release the acdbEntGet() list.
    acutRelRb(ebuf);
}
int printdxf(eb)
struct resbuf *eb;
{
    int rt;
    if (eb == NULL)
        return RTNONE;
    if ((eb->restype >= 0) && (eb->restype <= 9))
        rt = RTSTR ;
    else if ((eb->restype >= 10) && (eb->restype <= 19))
        rt = RT3DPOINT;
    else if ((eb->restype >= 38) && (eb->restype <= 59))
        rt = RTREAL ;
    else if ((eb->restype >= 60) && (eb->restype <= 79))
        rt = RTSHORT ;
    else if ((eb->restype >= 210) && (eb->restype <= 239))
        rt = RT3DPOINT ;
    else if (eb->restype < 0)
        // Entity name (or other sentinel)
        rt = eb->restype;
    else
        rt = RTNONE;
    switch (rt) {
    case RTSHORT:
        acutPrintf("(%d . %d)\n", eb->restype,
            eb->resval.rint);
        break;
    case RTREAL:
        acutPrintf("(%d . %0.3f)\n", eb->restype,
            eb->resval.rreal);
        break;
    case RTSTR:
        acutPrintf("(%d . \"%s\")\n", eb->restype,
            eb->resval.rstring);
        break;
    case RT3DPOINT:
        acutPrintf("(%d . %0.3f %0.3f %0.3f)\n",
            eb->restype,
            eb->resval.rpoint[X], eb->resval.rpoint[Y],
            eb->resval.rpoint[Z]);
        break;
    case RTNONE:
        acutPrintf("(%d . Unknown type)\n", eb->restype);
        break;
    case -1:
    case -2:
        // First block entity
        acutPrintf("(%d . <Entity name: %8lx>)\n",
            eb->restype, eb->resval.rlname[0]);
    }
```

In the next example, the following (default) conditions apply to the current drawing.

- The current layer is 0
- The current linetype is CONTINUOUS
- The current elevation is 0
- Entity handles are disabled

Also, the user has drawn a line with the following sequence of commands:

Command: **line**

Specify first point: **1,2**

Specify next point or [Undo]: **6,6**

Specify next point or [Undo]: ENTER

Then a call to `getlast()` would print the following (the name value will vary).

Results from acdbEntGet() of last entity:

(-1 . <Entity name: 60000014>)

(0 . "LINE")

(8 . "0")

(10 1.0 2.0 0.0)

(11 6.0 6.0 0.0)

(210 0.0 0.0 1.0)

Note The `printdxf()` function prints the output in the format of an AutoLISP association list, but the items are stored in a linked list of result buffers.

The result buffer at the start of the list (with a -1 sentinel code) contains the name of the entity that this list represents. The `acdbEntMod()` function uses it to identify the entity to be modified.

The codes for the components of the entity (stored in the `restype` field) are those used by DXF. As with DXF, the entity header items are returned only if they have values other than the default. Unlike DXF, optional entity definition fields are returned regardless of whether they equal their defaults. This simplifies processing; an application can always assume that these fields are present. Also unlike DXF, associated *X*, *Y*, and *Z* coordinates are returned as a single point variable (`resval.rpoint`), not as separate *X* (10), *Y* (20), and *Z* (30) groups. The `restype` value contains the group number of the *X* coordinate (in the range 10–19).

To find a group with a specific code, an application can traverse the list. The `entitem()` function shown here searches a result buffer list for a group of a specified type.

```
static struct resbuf *entitem(rchain, gcode)
struct resbuf *rchain;
int gcode;
{
    while ((rchain != NULL) && (rchain->restype != gcode))
        rchain = rchain->rbnext;
    return rchain;
}
```

If the DXF group code specified by the gcode argument is not present in the list (or if gcode is not a valid DXF group), entitem() "falls off the end" and returns NULL. Note that entitem() is equivalent to the AutoLISP function (assoc).

The acdbEntMod() function modifies an entity. It passes a list that has the same format as a list returned by acdbEntGet(), but with some of the entity group values (presumably) modified by the application. This function complements acdbEntGet(); the primary means by which an ObjectARX application updates the database is by retrieving an entity with acdbEntGet(), modifying its entity list, and then passing the list back to the database with acdbEntMod().

Note To restore the default value of an entity's color or linetype, use acdbEntMod() to set the color to 256, which is BYLAYER, or the linetype to BYLAYER.

The following code fragment retrieves the definition data of the first entity in the drawing, and changes its layer property to MYLAYER.

```
ads_name en;
struct resbuf *ed, *cb;
char *nl = "MYLAYER";
if (acdbEntNext(NULL, en) != RTNORM)
    return BAD; // Error status
ed = acdbEntGet(en); // Retrieve entity data.
for (cb = ed; cb != NULL; cb = cb->rbnext)
    if (cb->restype == 8) {
        // Check to make sure string buffer is long enough.
        if (strlen(cb->resval.rstring) < (strlen(nl)))
            // Allocate a new string buffer.
            cb->resval.rstring = realloc(cb->resval.rstring,
                strlen(nl) + 1);
        strcpy(cb->resval.rstring, nl);
        if (acdbEntMod(ed) != RTNORM) {
            acutRelRb(ed);
            return BAD; // Error
        }
        break; // From the for loop
    }
```

Memory management is the responsibility of an ObjectARX application. Code in the example ensures that the string buffer is the correct size, and it releases the result buffer returned by acdbEntGet() (and passed to acdbEntMod()) once the operation is completed, whether or not the call to acdbEntMod() succeeds.

Note If you use acdbEntMod() to modify an entity in a block definition, this affects all INSERT or XREF references to that block; also, entities in block definitions cannot be deleted by acdbEntDel().

An application can also add an entity to the drawing database by calling the `acdbEntMake()` function. Like `acdbEntMod()`, the argument to `acdbEntMake()` is a result-buffer list whose format is similar to that of a list returned by `acdbEntGet()`. (The `acdbEntMake()` call ignores the entity name field [-1] if that is present.) The new entity is appended to the drawing database (it becomes the last entity in the drawing). If the entity is a complex entity (a polyline or block), it is not appended to the database until it is complete.

The following sample code fragment creates a circle on the layer MYLAYER.

```
int status;
struct resbuf *entlist;
ads_point center = {5.0, 7.0, 0.0};
char *layer = "MYLAYER";
entlist = acutBuildList(RTDXF0, "CIRCLE",// Entity type
    8, layer, // Layer name
    10, center, // Center point
    40, 1.0, // Radius
    0 );
if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake buffer.
if (status == RTERROR) {
    acdbFail("Unable to make circle entity\n");
    return BAD;
}
```

Both `acdbEntMod()` and `acdbEntMake()` perform the same consistency checks on the entity data passed to them as the AutoCAD DXFIN command performs when reading DXF files. They fail if they cannot create valid drawing entities.

Topics in this section

- Complex Entities
- Anonymous Blocks

Complex Entities

A complex entity (a polyline or block) must be created by multiple calls to `acdbEntMake()`, using a separate call for each subentity. When `acdbEntMake()` first receives an initial component for a complex entity, it creates a temporary file in which to gather the definition data (and extended data, if present). Each subsequent `acdbEntMake()` call appends the new subentity to the file. When the definition of the complex entity is complete (that is, when `acdbEntMake()` receives an appropriate `Seqend` or `Endblk` subentity), the entity is checked for consistency, and if valid, it is added to the drawing. The file is deleted when the complex entity is complete or when its creation is canceled.

The following example contains five calls to `acdbEntMake()` that create a single complex entity, a polyline. The polyline has a linetype of DASHED and a color of BLUE. It has three vertices located at coordinates (1,1,0), (4,6,0), and (3,2,0). All other optional definition data assume default values.

```
int status;
struct resbuf *entlist, result;
ads_point newpt;
entlist = acutBuildList(
    RTDXF0, "POLYLINE",// Entity type
    62, 5, // Color (blue)
    6, "dashed",// Linetype
    66, 1, // Vertices follow.
    0);
if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake() buffer.
if (status != RTNORM) {
    acutPrintf ("%d",status);
    acedGetVar ("ERRNO", &result);
    acutPrintf ("ERRNO == %d, result.resval.rint);
    acdbFail("Unable to start polyline\n");
    return BAD;
}
newpt[X] = 1.0;
newpt[Y] = 1.0;
newpt[Z] = 0.0; // The polyline is planar
entlist = acutBuildList(
    RTDXF0, "VERTEX", // Entity type
    62, 5, // Color (blue)
    6, "dashed", // Linetype
    10, newpt, // Start point
    0);
if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake() buffer.
if (status != RTNORM) {
    acdbFail("Unable to add polyline vertex\n");
    return BAD;
}
newpt[X] = 4.0;
newpt[Y] = 6.0;
entlist = acutBuildList(
    RTDXF0, "VERTEX", // Entity type
    62, 5, // Color (blue)
    6, "dashed", // Linetype
    10, newpt, // Second point
    0);
if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake() buffer.
if (status != RTNORM) {
    acdbFail("Unable to add polyline vertex\n");
    return BAD;
}
newpt[X] = 3.0;
newpt[Y] = 2.0;
entlist = acutBuildList(
    RTDXF0, "VERTEX", // Entity type
```

Creating a block is similar, except that when `acdbEntMake()` successfully creates the `Endblk` entity, it returns a value of `RTKWORD`. You can verify the name of the new block by a call to `acedGetInput()`.

Anonymous Blocks

You can create anonymous blocks by calls to `acdbEntMake()`. To do so, you must open the block with a name whose first character is * and a block typeflag (group 70) whose low-order bit is set to 1. AutoCAD assigns the new anonymous block a name; characters in the name string that follow the * are often ignored. You then create the anonymous block the way you would create a regular block, except that it is more important to call `acedGetInput()`. Because the name is generated by AutoCAD, your program has no other way of knowing the name of the new block.

The following code begins an anonymous block, ends it, and retrieves its name.

```
int status;
struct resbuf *entlist;
ads_point basept;
char newblkname[20];
ads_point pnt1 = ( 0.0, 0.0, 0.0);
entlist = acutBuildList(
    RTDXF0, "BLOCK",
    2, "*ANON", // Only the '*' matters.
    10, "1", // No other flags are set.
    0 );
if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake buffer.
if (status != RTNORM) {
    acdbFail("Unable to start anonymous block\n");
    return BAD;
}
// Add entities to the block by more acdbEntMake calls.
.
.
.
entlist = acutBuildList(RTDXF0, "ENDBLK", 0 );
if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake buffer.
if (status != RTKWORD) {
    acdbFail("Unable to close anonymous block\n");
    return BAD;
}
status = acedGetInput(newblkname);
if (status != RTNORM) {
    acdbFail("Anonymous block not created\n");
    return BAD;
}
```

To reference an anonymous block, create an insert entity with `acdbEntMake()`. (You cannot pass an anonymous block to the INSERT command.)

Continuing the previous example, the following code fragment inserts the anonymous block at (0,0).

```
basept[X] = basept[Y] = basept[Z] = 0.0;
entlist = acutBuildList(
    RTDXF0, "INSERT",
    2, newblkname, // From acedGetInput
    10, basept,
    0 );
if (entlist == NULL) {
    acdbFail("Unable to create result buffer list\n");
    return BAD;
}
status = acdbEntMake(entlist);
acutRelRb(entlist); // Release acdbEntMake buffer.
if (status != RTNORM) {
    acdbFail("Unable to insert anonymous block\n");
    return BAD;
}
```

Entity Data Functions and Graphics Screen

Changes to the drawing made by the entity data functions are reflected on the graphics screen, provided that the entity being deleted, undeleted, modified, or made is in an area and is on a layer that is currently visible. There is one exception: when `acdbEntMod()` modifies a subentity, it does not update the image of the entire (complex) entity. The reason should be clear. If, for example, an application were to modify 100 vertices of a complex polyline with 100 iterated calls to `acdbEntMod()`, the time required to recalculate and redisplay the entire polyline as each vertex was changed would be unacceptably slow. Instead, an application can perform a series of subentity modifications and then redisplay the entire entity with a single call to the `acdbEntUpd()` function.

In the following example, the first entity in the current drawing is a polyline with several vertices. The following code modifies the second vertex of the polyline and then regenerates its screen image.

```
ads_name e1, e2;
struct resbuf *ed, *cb;
if (acdbEntNext(NULL, e1) != RTNORM) {
    acutPrintf("\nNo entities found.  Empty drawing.");
    return BAD;
}
acdbEntNext(e1, e2);
if ((ed = acdbEntGet(e2)) != NULL) {
    for (cb = ed; cb != NULL; cb = cb->rbnext)
        if (cb->restype == 10) { // Start point DXF code
            cb->resval.rpoint[X] = 1.0;// Change coordinates.
            cb->resval.rpoint[Y] = 2.0;
            if (acdbEntMod(ed) != RTNORM) { // Move vertex.
                acutPrintf("\nBad vertex modification.");
                acutRelRb(ed);
                return BAD;
            } else {
                acdbEntUpd(e1); // Regen the polyline.
                acutRelRb(ed);
                return GOOD; // Indicate success.
            }
        }
    acutRelRb(ed);
}
return BAD; // Indicate failure.
```

The argument to `acdbEntUpd()` can specify either a main entity or a subentity; in either case, `acdbEntUpd()` regenerates the entire entity. Although its primary use is for complex entities, `acdbEntUpd()` can regenerate any entity in the current drawing.

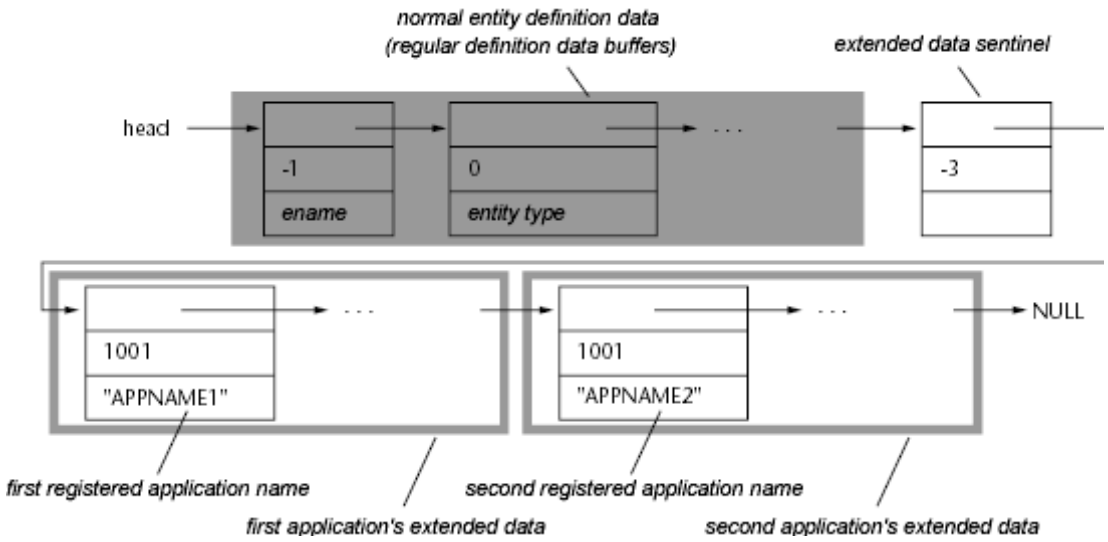
Note If the modified entity is in a block definition, then the `acdbEntUpd()` function is not sufficient. You must regenerate the drawing by invoking the AutoCAD REGEN command (with `acedCmd()` or `acedCommand()`) to ensure that all instances of the block references are updated.

Notes on Extended Data

Several ObjectARX functions are provided to handle extended data. An entity's extended data follows the entity's normal definition data. This is illustrated by the next figure, which shows the scheme of a result-buffer list for an entity containing extended data.

An entity's extended data can be retrieved by calling `acdbEntGetX()`, which is similar to `acdbEntGet()`. The `acdbEntGetX()` function retrieves an entity's normal definition data and the extended data for applications specified in the `acdbEntGetX()` call.

Note When extended data is retrieved with `acdbEntGetX()`, the beginning of extended data is indicated by a -3 sentinel code; the -3 sentinel is in a result buffer that precedes the first 1001 group. The 1001 group contains the application name of the first application retrieved, as shown in the figure.



Topics in this section

- Organization of Extended Data
- Registering an Application
- Retrieving Extended Data
- Managing Extended Data Memory Use
- Using Handles in Extended Data

Organization of Extended Data

Extended data consists of one or more 1001 groups, each of which begins with a unique application name. Application names are string values. The extended data groups returned by `acdbEntGetX()` follow the definition data in the order in which they are saved in the database.

Within each application's group, the contents, meaning, and organization of the data are defined by the application; AutoCAD maintains the information but doesn't use it. Group codes for extended data are in the range 1000–1071, as follows:

String

1000. Strings in extended data can be up to 255 bytes long (with the 256th byte reserved for the null character).

Application name

1001 (also a string value). Application names can be up to 31 bytes long (the 32nd byte is reserved for the null character) and must adhere to the rules for symbol table names (such as layer names). An application name can contain letters, digits, and the special characters $ (dollar sign), - (hyphen), and _ (underscore). It cannot contain spaces. Letters in the name are converted to uppercase.

A group of extended data cannot consist of an application name with no other data.

To delete extended data

1. Call `acdbEntGet()` to retrieve the entity.
2. Add to the end of the list returned by `acdbEntGet()` a resbuf with a `restype` of -3.
3. Add to the end of the list another resbuf with a `restype` of 1001 and a `resval.rstring` set to `<appname>`.

If you attempt to add a 1001 group but no other extended data to an existing entity, the attempt is ignored. If you attempt to make an entity whose only extended data group is a single 1001 group, the attempt fails.

Layer name

1003. Name of a layer associated with the extended data.

Database handle

1005. Handles of entities in the drawing database. Under certain conditions, AutoCAD translates these.

3D point

1010. Three real values, contained in a point.

Real

1040. A real value.

Integer

1070. A 16-bit integer (signed or unsigned).

Long

1071. A 32-bit signed (long) integer. If the value that appears in a 1071 group is a short integer or a real value, it is converted to a long integer; if it is invalid (for example, a string), it is converted to a long zero (0L).

Control string

1002. An extended data control string can be either "{" or "}". These braces enable the application to organize its data by subdividing it into lists. The left brace begins a list, and the right brace terminates the most recent list. (Lists can be nested.) When it reads the extended data, AutoCAD checks to ensure that braces are balanced correctly.

Binary data

1004. Binary data is organized into variable-length chunks, which can be handled in ObjectARX with the `ads_binary` structure. The maximum length of each chunk is 127 bytes.

World space position

1011. Unlike a simple 3D point, the world space coordinates are moved, scaled, rotated, and mirrored along with the parent entity to which the extended data belongs. The world space position is also stretched when the STRETCH command is applied to the parent entity and this point lies within the selection window.

World space displacement

1012. A 3D point that is scaled, rotated, or mirrored along with the parent, but not stretched or moved.

World direction

1013. Also a 3D point that is rotated, or mirrored along with the parent, but not scaled, stretched, or moved. The world direction is a normalized displacement that always has a unit length.

Distance

1041. A real value that is scaled along with the parent entity.

Scale factor

1042. Also a real value that is scaled along with the parent.

Note

If a 1001 group appears within a list, it is treated as a string and does not begin a new application group.

Registering an Application

Application names are saved with the extended data of each entity that uses them and in the APPID table. An application must register the name or names it uses. In ObjectARX, this is done by a call to `acdbRegApp()`. The `acdbRegApp()` function specifies a string to use as an application name. It returns `RTNORM` if it can successfully add the name to APPID; otherwise, it returns `RTERROR`. A result of `RTERROR` usually indicates that the name is already in the symbol table. This is not an actual error condition but a normally expected return value, because the application name needs to be registered only once per drawing.

To register itself, an application should first check that its name is not already in the APPID table, because `acdbRegApp()` needs to be called only once per drawing. If the name is not there, the application must register it; otherwise, it can go ahead and use the data.

The following sample code fragment shows the typical use of `acdbRegApp()`.

```
#define APPNAME "Local_Operation_App_3-2"
struct resbuf *rbp;
static char *local_appname = APPNAME;
// The static declaration prevents a copy being made of the string
// every time it's referenced.
.
.
.
if ((rbp = acdbTblSearch("APPID", local_appname, 0)) == NULL) {
    if (acdbRegApp(APPNAME) != RTNORM) { // Some other
                                         // problem
        acutPrintf("Can't register XDATA for %s.",
                   local_appname);
        return BAD;
    }
} else {
    acutRelRb(rbp);
}
```

Retrieving Extended Data

An application can obtain registered extended data by calling the acdbEntGetX() function, which is similar to acdbEntGet(). While acdbEntGet() returns only definition data, acdbEntGetX() returns both the definition data and the extended data for the applications it requests. It requires an additional argument, apps, that specifies the application names (this differs from AutoLISP, in which the (entget) function has been extended to accept an optional argument that specifies application names). The names passed to acdbEntGetX() must correspond to applications registered by a previous call to acdbRegApp(); they can also contain wild-card characters. If the apps argument is a NULL pointer, the call to acdbEntGetX() is identical to an acdbEntGet() call.

The following sample code fragment shows a typical sequence for retrieving extended data for two specified applications. Note that the apps argument passes application names in linked result buffers.

```
static struct resbuf    appname2 = {NULL, RTSTR},
                        appname1 = {&appname2, RTSTR},
                        *working_ent;
strsave(appname1.rstring, "MY_APP_1");
strsave(appname2.rstring, "SOMETHING_ELSE");
.
.
.
// Only extended data from "MY_APP_1" and
// "SOMETHING_ELSE" are retrieved:
working_ent = acdbEntGetX(&work_ent_addr, &appname1);
if (working_ent == NULL) {
    // Gracefully handle this failure.
    .
    .
    .
}
// Update working entity groups.
status = acdbEntMod(working_ent);
// Only extended data from registered applications still in the
// working_ent list are modified.
```

As the sample code shows, extended data retrieved by the acdbEntGetX() function can be modified by a subsequent call to acdbEntMod(), just as acdbEntMod() is used to modify normal definition data. (Extended data can also be created by defining it in the entity list passed to acdbEntMake().)

Returning the extended data of only specifically requested applications protects one application from damaging the data of another application. It also controls the amount of memory that an application uses, and simplifies the extended data processing that an application performs.

Note Because the strings passed with apps can include wild-card characters, an application name of "*" will cause acdbEntGetX() to return all extended data attached to an entity.

Managing Extended Data Memory Use

Extended data is limited to 16 kilobytes per entity. Because the extended data of an entity

can be created and maintained by multiple applications, this can lead to problems when the size of the extended data approaches its limit. ObjectARX provides two functions, `acdbXdSize()` and `acdbXdRoom()`, to assist in managing the memory that extended data occupies. When `acdbXdSize()` is passed a result-buffer list of extended data, it returns the amount of memory (in bytes) that the data will occupy; when `acdbXdRoom()` is passed the name of an entity, it returns the remaining number of free bytes that can still be appended to the entity.

The `acdbXdSize()` function must read an extended data list, which can be large. Consequently, this function can be slow, so it is recommended that you don't call it frequently. A better approach is to use it (in conjunction with `acdbXdRoom()`) in an error handler. If a call to `acdbEntMod()` fails, you can use `acdbXdSize()` and `acdbXdRoom()` to find out whether the call failed because the entity ran out of extended data, and then take appropriate action if that is the reason for the failure.

Using Handles in Extended Data

Extended data can contain handles (group 1005) to save relational structures within a drawing. One entity can reference another by saving the other entity's handle in its extended data. The handle can be retrieved later and passed to `acdbHandEnt()` to obtain the other entity. Because more than one entity can reference another, extended data handles are not necessarily unique; the AUDIT command does require that handles in extended data are either `NULL` or valid entity handles (within the current drawing). The best way to ensure that extended entity handles are valid is to obtain a referenced entity's handle directly from its definition data, by means of `acdbEntGet()`. (The handle value is in group 5 or 105.)

To reference entities in other drawings (for example, entities that are attached by means of an xref), you can avoid protests from AUDIT by using extended entity strings (group 1000) rather than handles (group 1005), because the handles of cross-referenced entities either are not valid in the current drawing or conflict with valid handles. However, if an XREF Attach changes to an XREF Bind or is combined with the current drawing in some other way, it is up to the application to revise entity references accordingly.

Note When drawings are combined by means of INSERT, INSERT *, XREF Bind (XBIND), or partial DXFIN, handles are translated so that they become valid in the current drawing. (If the incoming drawing did not employ handles, new ones are assigned.) Extended entity handles that refer to incoming entities are also translated when these commands are invoked.

When an entity is placed in a block definition (by means of the BLOCK command), the entity within the block is assigned new handles. (If the original entity is restored with OOPS, it retains its original handles.) The value of any extended data handles remains unchanged. When a block is exploded (with EXPLODE), extended data handles are translated, in a manner similar to the way they are translated when drawings are combined. If the extended data handle refers to an entity not within the block, it is unchanged; but if the extended data handle refers to an entity within the block, it is assigned the value of the new (exploded) entity's handle.

Xrecord Objects

The xrecord object is a built-in object class with a DXF name of "XRECORD", which stores and manages arbitrary data streams, represented externally as a result-buffer list composed of DXF groups with "normal object" groups (that is, non-xdata group codes), ranging from 1 through 369.

Xrecord objects are generic objects intended for use by ObjectARX and AutoLISP applications. This class allows applications to create and store arbitrary object structures of arbitrary result-buffer lists of non-graphical information completely separate from entities. The root owner for all application-defined objects is either the named object dictionary, which accepts any `AcDbObject` type as an entry, including `AcDbXrecord`, or the extension dictionary of any object.

Applications are expected to use unique entry names in the named object dictionary. The logic of using a named object dictionary or extension dictionary entry name is similar to that of a REGAPP name. In fact, REGAPP names are perfect for use as entry names when appending application-defined objects to the database or a particular object.

The use of xrecord objects represents a substantial streamlining with respect to the current practice of assigning xdata to entities. Because an xrecord object does not need to be linked with an entity, you no longer need to create dummy entities (dummy entities were often used to provide more room for xdata), or entities on frozen layers.

Applications can use xrecord objects to do the following:

- Protect information from indiscriminate purging or thawing of layers, which is always a threat to nongraphical information stored in xdata.
- Utilize the object ownership/pointer reference fields (330–369) to maintain internal database object references. Arbitrary handle values are completely exempt from the object ID translation mechanics. This is opposed to 1005 xdata groups, which are translated in some cases but not in others.
- Remain unaffected by the 16K per object xdata capacity limit. This object can also be used instead of xdata on specific entities and objects, if one so wishes, with the understanding that no matter where you store xrecord objects, they have no built-in size limit, other than the limit of 2 GB imposed by signed 32-bit integer range.

  In the case of object-specific state, xrecord objects are well suited for storing larger amounts of stored information, while xdata is better suited for smaller amounts of data.

When building up a hierarchy of xrecord objects (adding ownership or pointer reference to an object), that object must already exist in the database, and, thus, have a legitimate entity name. Because `acdbEntMake()` does not return an entity name, and `acdbEntLast()` only recognizes graphical objects, you must use `acdbEntMakeX()` if you are referencing nongraphical objects.

The `acdbEntMakeX()` function returns the entity name of the object added to the database (either graphical or nongraphical). If the object being created is an entity or a symbol table record, then `acdbEntMakeX()` will behave the same as `acdbEntMake()` (in other words, it will create the object and establish the object's ownership). For all other object types, `acdbEntMakeX()` appends the object to the database, but does not establish ownership.

Symbol Table Access

The `acdbTblNext()` function sequentially scans symbol table entries, and the `acdbTblSearch()` function retrieves specific entries. Table names are specified by strings.

The valid names are "LAYER", "LTYPE", "VIEW", "STYLE", "BLOCK", "UCS", "VPORT", and "APPID". Both of these functions return entries as result-buffer lists with DXF group codes.

The first call to acdbTblNext() returns the first entry in the specified table. Subsequent calls that specify the same table return successive entries unless the second argument to acdbTblNext() (rewind) is nonzero, in which case acdbTblNext() returns the first entry again.

In the following example, the function getblock() retrieves the first block (if any) in the current drawing, and calls the printdxf() function to display that block's contents in a list format.

```
void getblock()
{
    struct resbuf *bl, *rb;
    bl = acdbTblNext("BLOCK", 1); // First entry
    acutPrintf("\nResults from getblock():\n");
// Print items in the list as "assoc" items.
    for (rb = bl; rb != NULL; rb = rb->rbnext)
        printdxf(rb);
    // Release the acdbTblNext list.
    acutRelRb(bl);
}
```

Entries retrieved from the BLOCK table contain a -2 group that contains the name of the first entity in the block definition. In a drawing with a single block named BOX, a call to getblock() prints the following (the name value varies from session to session):

Results from getblock():

(0 . "BLOCK")

(2 . "BOX")

(70 . 0)

(10 9.0 2.0 0.0)

(-2 . <Entity name: 40000126>)

The first argument to acdbTblSearch() is a string that names a table, but the second argument is a string that names a particular symbol in the table. If the symbol is found, acdbTblSearch() returns its data. This function has a third argument, setnext, that can be used to coordinate operations with acdbTblNext(). If setnext is zero, the acdbTblSearch() call has no effect on acdbTblNext(), but if setnext is nonzero, the next call to acdbTblNext() returns the table entry that follows the entry found by acdbTblSearch().

The setnext option is especially useful when dealing with the VPORT symbol table, because all viewports in a particular viewport configuration have the same name (such as *ACTIVE).

Keep in mind that if the VPORT symbol table is accessed when TILEMODE is off, changes have no visible effect until TILEMODE is turned back on. (TILEMODE is set either by the SETVAR command or by entering its name directly.) Do not confuse the VPORT symbol table with viewport entities.

To find and process each viewport in the configuration named 4VIEW, you might use the following code:

```
struct resbuf *v, *rb;
v = acdbTblSearch("VPORT", "4VIEW", 1);
while (v != NULL} {
    for (rb = v; rb != NULL; rb = rb->rbnext)
        if (rb->restype == 2)
            if (strcmp(rb->resval.rstring, "4VIEW") == 0) {
                .// Process the VPORT entry
                .
                .
                acutRelRb(v);
                // Get the next table entry.
                v = acdbTblNext("VPORT", 0);
            } else {
                acutRelRb(v);
                v = NULL; // Break out of the while loop.
                break; // Break out of the for loop.
            }
}
```

Plot API

ObjectARX ® provides classes and functions that enable applications to control the plotting process, including what to plot and how to plot it.

Topics in this section

- Overview of the Plot API
- Using the Plot API

Overview of the Plot API

The ObjectARX plot API allows applications to determine what to plot, configure how to plot it, and generate AutoCAD ® plots based on those settings. The plot API classes fall into three groups: configuration-time classes, plot-time classes, and plot utility classes.

Configuration-time classes control what to plot and how to plot it. Configuration-time activities include selecting a device or PC3 file to use for a plot, determining the layout to plot, changing plot parameters (for example, plot origin, rotation, and scale) on the layout, providing overrides to these settings, selecting paper sizes, and so on.

Classes used to configure the plot include the following:

- AcDbLayout
- AcDbPlotSettings
- AcDbPlotSettingsValidator

- AcPlPlotInfo
- AcPlPlotInfoValidator
- AcPlPlotConfig
- AcPlPlotConfigInfo
- AcPlPlotConfigManager

Plot-time classes control generation of the plot using information from the configuration-time objects. Plot-time classes also support plot progress and notification.

Classes used to generate the plot include the following:

- AcPlPlotFactory
- AcPlPlotEngine
- AcPlPlotPageInfo
- AcPlPlotProgress
- AcPlPlotProgressDialog
- AcPlPlotReactor
- AcPlPlotReactorMgr

In addition to the configuration-time and plot-time subcomponents, utility classes support plot error handling, reading information from DSD files, and developing host applications that support plotting services.

Plot utility classes include the following:

- AcPlPlotErrorHandler
- AcPlPlotLoggingErrorHandler
- AcPlPlotErrorHandlerLock
- AcPlPlotLogger
- AcPlDSDData
- AcPlDSDEntry
- AcPlHostAppServices
- AcPlObject

Topics in this section

- Plot API Terminology
- Plot Settings Validation

Plot API Terminology

Following are definitions of commonly used plot API terms.

Document

One or more pages to be plotted.

Document compatible

Two `AcPlPlotInfo` objects are said to be "document compatible" if they satisfy the following requirements:

- Both objects are validated.
- Their validated settings have the same device (PC3 file or system printer name).
- Their validated settings have the same page size.
- Their validated settings have the same orientation.

Page

A single layout or sheet to be plotted.

Plot Settings Validation

Plot settings are represented as `AcDbPlotSettings` objects, which are stored in several ways in an `AcDbDatabase`. One group of plot settings is stored with each layout in the database; these are the defaults that AutoCAD displays when a user edits layout properties in the AutoCAD Page Setup dialog box. Plot settings are also represented in an `AcDbDatabase` as named page setups that can be applied to override the plot settings stored in a layout.

Developers can create plot settings objects and use them as overrides when plotting layouts using the plot APIs. If you choose to do so, you must provide complete settings. In other words, the object must contain all the necessary information for plotting, and not just the desired overrides. To accomplish this, you can copy the plot settings from the layout and then change specific settings programmatically.

When a plot is initiated, an `AcPlPlotInfo` object is created. This object specifies which layout to plot and what settings to use. (The plot settings in the layout are used if the plot info object does not specify plot settings overrides.) A device override may also be applied by the user to make one-time changes, such as specifying double-sided copies or plotting to a DWF file.

Before plotting can begin, the `AcPlPlotInfo` object must be validated to ensure that the devices, media, plot style tables, and other plot specifications exist on the system and are available for the current plot. See the validate documentation for more information on specific checks that are performed during validation.

Using the Plot API

Applications may choose to use the plot API configuration-time capability, the plot-time capability, or both. They can also include provisions for multi-page documents, plotting to a file, error handling, and overriding the default plot progress dialog box.

Note

Any time the plot engine is accessed through the plot API, the system will plot in the foreground if the BACKGROUNDPLOT system variable is set to 0, and in the background if the BACKGROUNDPLOT system variable is set to 1,2, or 3.

To use the basic configuration-time and plot-time APIs

1. Get the ID of the layout and make it active in the editor. (The layout must be active by the time `beginPage()` is called in step 7.)
2. Create an `AcPlPlotInfo` object and set the layout ID on it.
3. Implement a UI to create an `AcDbPlotSettings` object (optional).
4. Pass the `AcPlPlotInfo` object a pointer to the plot settings object (if there is one) using `AcPlPlotInfo::setOverrideSettings()`.
5. Call `AcPlPlotInfoValidator::validate()` on the plot info object.

   This verifies that the overrides are compatible with the layout settings. For example, paper space and model space settings are mutually exclusive and the specified device must exist.
6. Create an `AcPlPlotEngine` object using `AcPlPlotFactory::createPreviewEngine()` or `AcPlPlotFactory::createPublishEngine()`.
7. Call the following functions on the engine:
8. `beginPlot()`
9. `beginDocument()`
10. `beginPage()`
11. `beginGenerateGraphics()`
12. `endGenerateGraphics()`
13. `endPage()`
14. `endDocument()`
15. `endPlot()`

The following code illustrates a basic use of the API and the default plot progress dialog box, which is discussed further in Plot Progress Dialog Box.

```
AcPlPlotEngine* pEngine = NULL;
if(Acad::eOk==AcPlPlotFactory::createPublishEngine(pEngine))
{
    // Here is the progress dialog for the current plot process...
    AcPlPlotProgressDialog *pPlotProgDlg=acplCreatePlotProgressDialog(
        acedGetAcadFrame()->m_hWnd,false,1);
    pPlotProgDlg->setPlotMsgString(
        AcPlPlotProgressDialog::PlotMSGIndex::kDialogTitle,
        "Plot API Progress");
    pPlotProgDlg->setPlotMsgString(
        AcPlPlotProgressDialog::PlotMSGIndex::kCancelJobBtnMsg,
        "Cancel Job");
    pPlotProgDlg->setPlotMsgString(
        AcPlPlotProgressDialog::PlotMSGIndex::kCancelSheetBtnMsg,
        "Cancel Sheet");
    pPlotProgDlg->setPlotMsgString(
        AcPlPlotProgressDialog::PlotMSGIndex::kSheetSetProgressCaption,
        "Job Progress");
    pPlotProgDlg->setPlotMsgString(
        AcPlPlotProgressDialog::PlotMSGIndex::kSheetProgressCaption,
        "Sheet Progress");
    pPlotProgDlg->setPlotProgressRange(0,100);
    pPlotProgDlg->onBeginPlot();
    pPlotProgDlg->setIsVisible(true);
    es = pEngine->beginPlot(pPlotProgDlg);
    AcPlPlotPageInfo pageInfo;
    // Used to describe how the plot is to be made.
    AcPlPlotInfo plotInfo;
    // First, set the layout to the specified layout
    // (which is the current layout in this sample).
    plotInfo.setLayout(layoutId);// This is required.
    // Now, override the layout settings with the plot settings
    // we have been populating.
    plotInfo.setOverrideSettings(pPlotSettings);
    // We need to validate these settings.
    AcPlPlotInfoValidator validator;
    validator.setMediaMatchingPolicy(
        AcPlPlotInfoValidator::MatchingPolicy::kMatchEnabled);
    es = validator.validate(plotInfo);
    // Begin document.  The version we call is dependent
    // on the plot-to-file status.
    const char *szDocName=acDocManager->curDocument()->fileName();
    if(m_bPlotToFile)
        es = pEngine->beginDocument(plotInfo, szDocName,
            NULL, 1, true, m_csFilename);
    else
        es = pEngine->beginDocument(plotInfo, szDocName);
    // Follow through sending commands to the engine,
    // and notifications to the progress dialog.
    pPlotProgDlg->onBeginSheet();
    pPlotProgDlg->setSheetProgressRange(0, 100);
    pPlotProgDlg->setSheetProgressPos(0);
    es = pEngine->beginPage(pageInfo, plotInfo, true);
    es = pEngine->beginGenerateGraphics();
    es = pEngine->endGenerateGraphics();
    es = pEngine->endPage();
    pPlotProgDlg->setSheetProgressPos(100);
    pPlotProgDlg->onEndSheet();
    pPlotProgDlg->setPlotProgressPos(100);
    es = pEngine->endDocument();
    es = pEngine->endPlot();
    // Destroy the engine
    pEngine->destroy();
```

Topics in this section

- Multi-page Documents
- Plot-to-File
- Error Handling
- Plot Progress Dialog Box

Multi-page Documents

In order to support plotting multi-page documents, an application may nest multiple `AcPlPlotEngine::beginPage()/endPage()` pairs between the `beginDocument()` and `endDocument()` calls. For example:

```
beginPlot(progress);
    beginDocument(infoObj, myDoc);
        beginPage(pgInfo, infoObj, false);
            beginGenerateGraphics();
            endGenerateGraphics();
        endPage();
        beginPage(pgInfo2, infoObj, true);
            beginGenerateGraphics();
            endGenerateGraphics();
        endPage();
    endDocument();
endPlot();
```

The following restrictions apply when plotting multi-page documents:

- The device must support multi-page capability.
- All the pages in the job must be document compatible. (See Plot API Terminology for a definition of the term.)

  If an `AcPlPlotInfo` object passed to `beginPage()` is not document compatible with the `AcPlPlotInfo` object passed to `beginDocument()`, `beginPage()` returns an error.
- When calling `beginPage()`, applications must set the `bLastPage` argument to false for every page except the last one.

Plot-to-File

Applications can set a document to plot to a file by calling `AcPlPlotEngine::beginDocument()` with the `bPlotToFile` argument set to true and `pFileName` containing a fully qualified file name. In order to plot to a file, the selected device must support that capability.

The following code shows plotting to a file, with plot progress dialog code omitted for clarity:

```
AcPlPlotEngine* pEngine = NULL;
if(Acad::eOk==AcPlPlotFactory::createPublishEngine(pEngine))
{
    ...
    es = pEngine->beginPlot(pPlotProgDlg);
    AcPlPlotPageInfo pageInfo;
    AcPlPlotInfo plotInfo;
    plotInfo.setLayout(layoutId);
    AcPlPlotInfoValidator validator;
    es = validator.validate(plotInfo);
    const char *szDocName=acDocManager->curDocument()->fileName();
    // Set bPlotToFile parameter to true.
    es = pEngine->beginDocument(plotInfo, szDocName,
            NULL, 1, true, m_csFilename);
    es = pEngine->beginPage(pageInfo, plotInfo, true);
    es = pEngine->beginGenerateGraphics();
    es = pEngine->endGenerateGraphics();
    es = pEngine->endPage();
    es = pEngine->endDocument();
    es = pEngine->endPlot();
    // Destroy the engine.
    pEngine->destroy();
    pEngine = NULL;
else
    // Ensure the engine is not already busy...
    AfxMessageBox("Plot Engine is Busy...");
}
```

Error Handling

The plot APIs allow applications to determine how the host application reacts if various types of errors occur during plotting. To add an error handler to the error handling chain, applications should implement the AcPlPlotErrorHandler interface.

Plot Progress Dialog Box

`AcPlPlotEngine` does not display a plot progress dialog box automatically. Applications can instantiate the default implementation of the plot progress dialog box using the global function `acplCreatePlotProgressDialog()` and passing it to the plot engine. When this object is no longer needed, the caller is responsible for destroying it using `AcPlPlotProgressDialog::destroy()`.

Applications can customize the appearance of the plot progress dialog box using `AcPlPlotProgressDialog::setPlotMsgString()`. Applications also can provide their own plot progress dialog box by implementing the `AcPlPlotProgress` interface.

The following sample demonstrates how to instantiate the default implementation of the Plot Progress dialog box and pass it to the plot engine.

```
AcPlPlotProgressDialog *pPlotProgDlg=acplCreatePlotProgressDialog(
    acedGetAcadFrame()->m_hWnd,false,1);
pPlotProgDlg->setPlotMsgString(
    AcPlPlotProgressDialog::PlotMSGIndex::kDialogTitle,
    "Plot API Progress");
pPlotProgDlg->setPlotMsgString(
    AcPlPlotProgressDialog::PlotMSGIndex::kCancelJobBtnMsg,
    "Cancel Job");
pPlotProgDlg->setPlotMsgString(
    AcPlPlotProgressDialog::PlotMSGIndex::kCancelSheetBtnMsg,
    "Cancel Sheet");
pPlotProgDlg->setPlotMsgString(
    AcPlPlotProgressDialog::PlotMSGIndex::kSheetSetProgressCaption,
    "Job Progress");
pPlotProgDlg->setPlotMsgString(
    AcPlPlotProgressDialog::PlotMSGIndex::kSheetProgressCaption,
    "Sheet Progress");
pPlotProgDlg->setPlotProgressRange(0,100);
pPlotProgDlg->onBeginPlot();
pPlotProgDlg->setIsVisible(true);
es = pEngine->beginPlot(pPlotProgDlg);
```

Global Functions for Interacting with AutoCAD

The global functions described in this section allow your application to communicate with AutoCAD ® . This section discusses functions for registering commands with AutoCAD, handling user input, handling data conversions, and setting up external devices such as the tablet.

Topics in this section

- AutoCAD Queries and Commands
- Getting User Input
- Conversions
- Character Type Handling
- Coordinate System Transformations
- Display Control
- Tablet Calibration
- Wild-Card Matching

AutoCAD Queries and Commands

The functions described in this section access AutoCAD commands and services.

Topics in this section

- General Access

General Access

The most general of the functions that access AutoCAD are acedCommand() and acedCmd(). Like the (command) function in AutoLISP, these functions send commands and other input directly to the AutoCAD Command prompt.

```
int

acedCommand(int rtype, ...);

int

acedCmd(const struct resbuf * rbp);
```

Unlike most other AutoCAD interaction functions, acedCommand() has a variable-length argument list: arguments to acedCommand() are treated as pairs except for RTLE and RTLB, which are needed to pass a pick point. The first of each argument pair identifies the result type of the argument that follows, and the second contains the actual data. The final argument in the list is a single argument whose value is either 0 or RTNONE. Typically, the first argument to acedCommand() is the type code RTSTR, and the second data argument is a string that is the name of the command to invoke. Succeeding argument pairs specify options or data that the specified command requires. The type codes in the acedCommand() argument list are result types.

The data arguments must correspond to the data types and values expected by that command's prompt sequence. These can be strings, real values, integers, points, entity names, or selection set names. Data such as angles, distances, and points can be passed either as strings (as the user might enter them) or as the values themselves (that is, as integer, real, or point values). An empty string (“”) is equivalent to entering a space on the keyboard.

Because of the type identifiers, the acedCommand() argument list is not the same as the argument list for the AutoLISP ® (command) routine. Be aware of this if you convert an AutoLISP routine into an ObjectARX ® application.

There are restrictions on the commands that acedCommand() can invoke, which are comparable to the restrictions on the AutoLISP (command) function.

Note The acedCommand() and acedCmd() functions can invoke the AutoCAD SAVE or SAVEAS command. When they do so, AutoLISP issues a kSaveMsg message to all other ObjectARX applications currently loaded, but not to the application that invoked SAVE. The comparable code is sent when these functions invoke NEW, OPEN, END, or QUIT from an application.

The following sample function shows a few calls to acedCommand().

```
int docmd()
{
    ads_point p1;
    ads_real rad;
    if (acedCommand(RTSTR, "circle", RTSTR, "0,0", RTSTR,
        "3,3", 0) != RTNORM)
        return BAD;
    if (acedCommand(RTSTR, "setvar", RTSTR, "thickness",
        RTSHORT, 1, 0) != RTNORM)
        return BAD;
    p1[X] = 1.0; p1[Y] = 1.0; p1[Z] = 3.0;
    rad = 4.5;
    if (acedCommand(RTSTR, "circle", RT3DPOINT, p1, RTREAL,
        rad, 0) != RTNORM)
        return BAD;
    return GOOD;
}
```

Provided that AutoCAD is at the Command prompt when this function is called, AutoCAD performs the following actions:

1. Draws a circle that passes through (3.0,3.0) and whose center is at (0.0,0.0).
2. Changes the current thickness to 1.0. Note that the first call to `acedCommand()` passes the points as strings, while the second passes a `short` integer. Either method is possible.
3. Draws another (extruded) circle whose center is at (1.0,1.0,3.0) and whose radius is 4.5. This last call to `acedCommand()` uses a 3D point and a real (double-precision floating-point) value. Note that points are passed by reference, because `ads_point` is an array type.

Topics in this section

- Using acedCmd()
- Pausing for User Input
- Passing Pick Points to AutoCAD Commands
- System Variables
- AutoLISP Symbols
- File Search
- Object Snap
- Viewport Descriptors
- Geometric Utilities
- The Text Box Utility Function

Using acedCmd()

The `acedCmd()` function is equivalent to `acedCommand()` but passes values to AutoCAD in the form of a result-buffer list. This is useful in situations where complex logic is involved in constructing a list of AutoCAD commands. The `acutBuildList()` function is useful for constructing command lists.

The acedCmd() function also has the advantage that the command list can be modified at runtime rather than be fixed at compile time. Its disadvantage is that it takes slightly longer to execute. For more information, see the *ObjectARX Reference*.

The following sample code fragment causes AutoCAD to perform a REDRAW on the current graphics screen (or viewport).

```
struct resbuf *cmdlist;
cmdlist = acutBuildList(RTSTR, "redraw", 0);
if (cmdlist == NULL) {
acdbFail("Couldn't create list\n");
return BAD;
}
acedCmd(cmdlist);
acutRelRb(cmdlist);
```

Pausing for User Input

If an AutoCAD command is in progress and AutoCAD encounters the PAUSE symbol as an argument to acedCommand() or acedCmd(), the command is suspended to allow direct user input, including dragging. The PAUSE symbol consists of a string that contains a single backslash. This is similar to the backslash pause mechanism provided for menus.

The following call to acedCommand() invokes the ZOOM command and then uses the PAUSE symbol so that the user can select one of the ZOOM options.

```
result = acedCommand(RTSTR, "Zoom", RTSTR, PAUSE, RTNONE);
```

The following call starts the CIRCLE command, sets the center point as (5,5), and then pauses to let the user drag the circle's radius on the screen. When the user specifies the chosen point (or enters the chosen radius), the function resumes, drawing a line from (5,5) to (7,5).

```
result = acedCommand(RTSTR, "circle", RTSTR, "5,5",
    RTSTR, PAUSE, RTSTR, "line", RTSTR, "5,5", RTSTR,
    "7,5", RTSTR, "", 0);
```

Passing Pick Points to AutoCAD Commands

Some AutoCAD commands (such as TRIM, EXTEND, and FILLET) require users to specify a pick point as well as the entity. To pass such pairs of entity and point data by means of acedCommand(), you must specify the name of the entity first and enclose the pair in the RTLB and RTLE result type codes.

The following sample code fragment creates a circle centered at (5,5) and a line that extends from (1,5) to (8,5); it assumes that the circle and line are created in an empty drawing. It then uses a pick point with the TRIM command to trim the line at the circle's

edge. The `acdbEntNext()` function finds the next entity in the drawing, and the `acdbEntLast()` function finds the last entity in the drawing.

```
ads_point p1;
ads_name first, last;
acedCommand(RTSTR, "Circle", RTSTR, "5,5", RTSTR, "2",
    0);
acedCommand(RTSTR, "Line", RTSTR, "1,5", RTSTR, "8,5",
    RTSTR, "", 0);
acdbEntNext(NULL, first);     // Get circle.
acdbEntLast(last);     // Get line.
// Set pick point.
p1[X] = 2.0;
p1[Y] = 5.0;
p1[Z] = 0.0;
acedCommand(RTSTR, "Trim", RTENAME, first, RTSTR, "",
    RTLB, RTENAME, last, RTPOINT, p1, RTLE,
    RTSTR, "", 0);
```

System Variables

A pair of functions, `acedGetVar()` and `acedSetVar()`, enable ObjectARX applications to inspect and change the value of AutoCAD system variables. These functions use a string to specify the variable name (in either uppercase or lowercase), and a (single) result buffer for the type and value of the variable. A result buffer is required in this case because the AutoCAD system variables come in a variety of types: integers, real values, strings, 2D points, and 3D points.

The following sample code fragment ensures that subsequent FILLET commands use a radius of at least 1.

```
struct resbuf rb, rb1;
acedGetVar("FILLETRAD", &rb);
rb1.restype = RTREAL;
rb1.resval.rreal = 1.0;
if (rb.resval.rreal < 1.0)
    if (acedSetVar("FILLETRAD", &rb1) != RTNORM)
        return BAD; // Setvar failed.
```

In this example, the result buffer is allocated as an automatic variable when it is declared in the application. The application does not have to explicitly manage the buffer's memory use as it does with dynamically allocated buffers.

If the AutoCAD system variable is a string type, `acedGetVar()` allocates space for the string. The application is responsible for freeing this space. You can do this by calling the standard C library function `free()`, as shown in the following example:

```
acedGetVar("TEXTSTYLE", &rb);
if (rb.resval.rstring != NULL)
    // Release memory acquired for string:
    free(rb.resval.rstring);
```

AutoLISP Symbols

The functions `acedGetSym()` and `acedPutSym()` let ObjectARX applications inspect and change the value of AutoLISP variables.

In the first example, the user enters the following AutoLISP expressions:

Command: **(setq testboole t)**

T

Command: **(setq teststr "HELLO, WORLD")**

"HELLO, WORLD"

Command: **(setq sset1 (ssget))**

<Selection set: 1>

Then the following sample code shows how `acedGetSym()` retrieves the new values of the symbols.

```
struct resbuf *rb;
int rc;
long sslen;
rc = acedGetSym("testboole", &rb);
if (rc == RTNORM && rb->restype == RTT)
    acutPrintf("TESTBOOLE is TRUE\n");
acutRelRb(rb);
rc = acedGetSym("teststr", &rb);
if (rc == RTNORM && rb->restype == RTSTR)
    acutPrintf("TESTSTR is %s\n", rb->resval.rstring);
acutRelRb(rb);
rc = acedGetSym("sset1", &rb);
if (rc == RTNORM && rb->restype == RTPICKS) {
    rc = acedSSLength(rb->resval.rlname, &sslen);
    acutPrintf("SSET1 contains %lu entities\n", sslen);
}
acutRelRb(rb);
```

Conversely, `acedPutSym()` can create or change the binding of AutoLISP symbols, as follows:

```
ads_point pt1;
pt1[X] = pt1[Y] = 1.4; pt1[Z] = 10.9923;
rb = acutBuildList(RTSTR, "GREETINGS", 0);
rc = acedPutSym("teststr", rb);
acedPrompt("TESTSTR has been reset\n");
acutRelRb(rb);
rb = acutBuildList(RTLB, RTSHORT, -1,
    RTSTR, "The combinations of the world",
    RTSTR, "are unstable by nature.", RTSHORT, 100,
    RT3DPOINT, pt1,
    RTLB, RTSTR, "He jests at scars",
    RTSTR, "that never felt a wound.", RTLE, RTLE, 0);
rc = acedPutSym("longlist", rb);
acedPrompt("LONGLIST has been created\n");
acutRelRb(rb);
```

To set an AutoLISP variable to nil, make the following assignment and function call:

```
rb->restype = RTNIL;
acedPutSym("var1", rb);
```

Users can retrieve these new values. (As shown in the example, your program should notify users of any changes.)

TESTSTR has been reset.

LONGLIST has been created.

Command: **!teststr**

("GREETINGS")

Command: **!longlist**

((-1 "The combinations of the world" "are unstable by nature." 100 (1.4 1.4 10.9923) ("He jests at scars" "that never felt a wound.")))

File Search

The `acedFindFile()` function enables an application to search for a file of a particular name. The application can specify the directory to search, or it can use the current AutoCAD library path.

In the following sample code fragment, `acedFindFile()` searches for the requested file name according to the AutoCAD library path.

```
char *refname = "refc.dwg";
char fullpath[100];
.
.
.
if (acedFindFile(refname, fullpath) != RTNORM) {
    acutPrintf("Could not find file %s.\n", refname);
    return BAD;
```

If the call to `acedFindFile()` is successful, the fullpath argument is set to a fully qualified path name string, such as the following:

```
/home/work/ref/refc.dwg
```

You can also prompt users to enter a file name by means of the standard AutoCAD file dialog box. To display the file dialog box, call `acedGetFileD()`.

The following sample code fragment uses the file dialog box to prompt users for the name of an ObjectARX application.

```
struct resbuf *result;
int rc, flags;
if (result = acutNewRb(RTSTR) == NULL) {
    acdbFail("Unable to allocate buffer\n");
    return BAD;
}
result->resval.rstring=NULL;
flags = 2; // Disable the "Type it" button.
rc = acedGetFileD("Get ObjectARX Application", // Title
    "/home/work/ref/myapp",   // Default pathname
    NULL, // The default extension: NULL means "*".
    flags, // The control flags
    result); // The path selected by the user.
if (rc == RTNORM)
    rc = acedArxLoad(result->resval.rstring);
```

Object Snap

The `acedOsnap()` function finds a point by using one of the AutoCAD Object Snap modes. The snap modes are specified in a string argument.

In the following example, the call to `acedOsnap()` looks for the midpoint of a line near `pt1`.

```
acedOsnap(pt1, "midp", pt2);
```

The following call looks for either the midpoint or endpoint of a line, or the center of an arc

or circle—whichever is nearest pt1.

```
acedOsnap(pt1, "midp,endp,center", pt2);
```

The third argument (pt2 in the examples) is set to the snap point if one is found. The acedOsnap() function returns RTNORM if a point is found.

Note

The APERTURE system variable determines the allowable proximity of a selected point to an entity when using Object Snap.

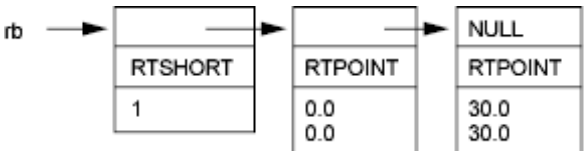
Viewport Descriptors

The function acedVports(), like the AutoLISP function (vports), gets a list of descriptors of the current viewports and their locations.

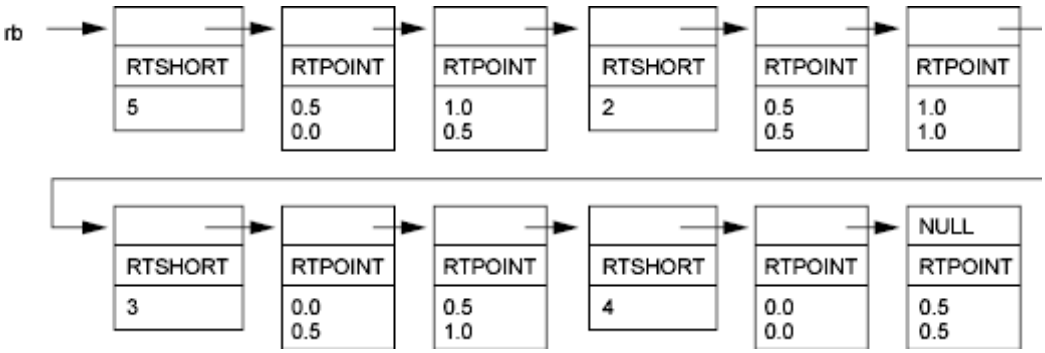
The following sample code gets the current viewport configuration and passes it back to AutoLISP for display.

```
struct resbuf *rb;
int rc;
rc = acedVports(&rb);
acedRetList(rb);
acutRelRb(rb);
```

For example, given a single-viewport configuration with TILEMODE turned on, the preceding code may return the list shown in the following figure.



Similarly, if four equal-sized viewports are located in the four corners of the screen and TILEMODE is turned on, the preceding code may return the configuration shown in the next figure.



The current viewport's descriptor is always first in the list. In the list shown in the preceding figure, viewport number 5 is the current viewport.

Geometric Utilities

One group of functions enables applications to obtain geometric information. The `acutDistance()` function finds the distance between two points, `acutAngle()` finds the angle between a line and the *X* axis of the current UCS (in the *XY* plane), and `acutPolar()` finds a point by means of polar coordinates (relative to an initial point). Unlike most ObjectARX functions, these functions do not return a status value. The `acdbInters()` function finds the intersection of two lines; it returns `RTNORM` if it finds a point that matches the specification.

Note Unlike `acedOsnap()`, the functions in this group simply calculate the point, line, or angle values, and do not actually query the current drawing.

The following sample code fragment shows some simple calls to the geometric utility functions.

```
ads_point pt1, pt2;
ads_point base, endpt;
ads_real rads, length;
.
.     // Initialize pt1 and pt2.
.
// Return the angle in the XY plane of the current UCS, in radians.
rads = acutAngle(pt1, pt2);
// Return distance in 3D space.
length = acutDistance(pt1, pt2);
base[X] = 1.0; base[Y] = 7.0; base[Z] = 0.0;
acutPolar(base, rads, length, endpt);
```

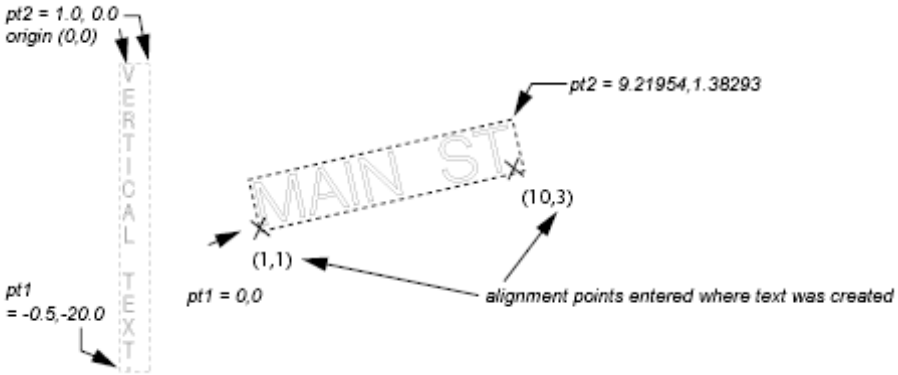
The call to `acutPolar()` sets `endpt` to a point that is the same distance from (1,7) as `pt1` is from `pt2`, and that is at the same angle from the *X* axis as the angle between `pt1` and `pt2`.

The Text Box Utility Function

The function `acedTextBox()` finds the diagonal coordinates of a box that encloses a text entity. The function takes an argument, `ent`, that must specify a text definition or a string group in the form of a result-buffer list. The `acedTextBox()` function sets its `p1` argument to the minimum *XY* coordinates of the box and its `p2` argument to the maximum *XY* coordinates.

If the text is horizontal and is not rotated, `p1` (the bottom-left corner) and `p2` (the top-right corner) describe the bounding box of the text. The coordinates are expressed in the Entity Coordinate System (ECS) of `ent` with the origin (0,0) at the left endpoint of the baseline. (The origin is not the bottom-left corner if the text contains letters with descenders, such as g and p.) For example, the following figure shows the results of applying `acedTextBox()` to a text entity with a height of 1.0. The figure also shows the baseline and origin of the text.

The next figure shows the point values that `acedTextBox()` returns for samples of vertical and aligned text. In both samples, the height of the letters was entered as 1.0. (For the rotated text, this height is scaled to fit the alignment points.)



Note that with vertical text styles, the points are still returned in left-to-right, bottom-to-top order, so the first point list contains negative offsets from the text origin.

The `acedTextBox()` function can also measure strings in attdef and attrib entities. For an attdef, `acedTextBox()` measures the tag string (group 2); for an attrib entity, it measures the current value (group 1).

The following function, which uses some entity handling functions, prompts the user to select a text entity, and then draws a bounding box around the text from the coordinates returned by `acedTextBox()`.

Note The sample `tbox()` function works correctly only if you are currently in the World Coordinate System (WCS). If you are not, the code should convert the ECS points retrieved from the entity into the UCS coordinates used by `acedCommand()`. See Coordinate System Transformations.

```
int tbox()
{
    ads_name tname;
    struct resbuf *textent, *tent;
    ads_point origin, lowleft, upright, p1, p2, p3, p4;
    ads_real rotatn;
    char rotatstr[15];
    if (acedEntSel("\nSelect text: ", tname, p1) != RTNORM) {
        acdbFail("No Text entity selected\n");
        return BAD;
    }
    textent = acdbEntGet(tname);
    if (textent == NULL) {
        acdbFail("Couldn't retrieve Text entity\n");
        return BAD;
    }
    tent = entitem(textent, 10);
    origin[X] = tent->resval.rpoint[X]; //ECS coordinates
    origin[Y] = tent->resval.rpoint[Y];
    tent = entitem(textent, 50);
    rotatn = tent->resval.rreal;
             // acdbAngToS() converts from radians to degrees.
    if (acdbAngToS(rotatn, 0, 8, rotatstr) != RTNORM) {
        acdbFail("Couldn't retrieve or convert angle\n");
        acutRelRb(textent);
        return BAD;
    }
    if (acedTextBox(textent, lowleft, upright) != RTNORM) {
        acdbFail("Couldn't retrieve text box
                  coordinates\n");
        acutRelRb(textent);
        return BAD;
    }
    acutRelRb(textent);
        // If not currently in the WCS, at this point add
        // acedTrans() calls to convert the coordinates
        // retrieved from acedTextBox().
    p1[X] = origin[X] + lowleft[X]; // UCS coordinates
    p1[Y] = origin[Y] + lowleft[Y];
    p2[X] = origin[X] + upright[X];
    p2[Y] = origin[Y] + lowleft[Y];
    p3[X] = origin[X] + upright[X];
    p3[Y] = origin[Y] + upright[Y];
    p4[X] = origin[X] + lowleft[X];
    p4[Y] = origin[Y] + upright[Y];
    if (acedCommand(RTSTR, "pline", RTPOINT, p1,
        RTPOINT, p2, RTPOINT, p3,RTPOINT, p4, RTSTR, "c",
        0) != RTNORM) {
            acdbFail("Problem creating polyline\n");
            return BAD;
    }
    if (acedCommand(RTSTR, "rotate", RTSTR, "L", RTSTR, "",
        RTPOINT, origin, RTSTR, rotatstr, 0) != RTNORM) {
            acdbFail("Problem rotating polyline\n");
            return BAD;
    }
    return GOOD;
}
```

The preceding example “cheats” by using the AutoCAD ROTATE command to cause the rotation. A more direct way to do this is to incorporate the rotation into the calculation of the box points, as follows:

```
ads_real srot, crot;
tent = entitem(textent, 50);
rotatn = tent->resval.rreal;
srot = sin(rotatn);
crot = cos(rotatn);
        .
        .
        .
p1[X] = origin[X] + (lowleft[X]*crot - lowleft[Y]*srot);
p1[Y] = origin[Y] + (lowleft[X]*srot + lowleft[Y]*crot);
p2[X] = origin[X] + (upright[X]*crot - lowleft[Y]*srot);
p2[Y] = origin[Y] + (upright[X]*srot + lowleft[Y]*crot);
p3[X] = origin[X] + (upright[X]*crot - upright[Y]*srot);
p3[Y] = origin[Y] + (upright[X]*srot + upright[Y]*crot);
p4[X] = origin[X] + (lowleft[X]*crot - upright[Y]*srot);
p4[Y] = origin[Y] + (lowleft[X]*srot + upright[Y]*crot);
```

Getting User Input

Several global functions enable an ObjectARX application to request data interactively from the AutoCAD user.

Topics in this section

- User-Input Functions
- Control of User-Input Function Conditions
- Graphically Dragging Selection Sets
- User Breaks
- Returning Values to AutoLISP Functions

User-Input Functions

The user-input or `acedGetxxx()` functions pause for the user to enter data of the indicated type, and return the value in a result argument. The application can specify an optional prompt to display before the function pauses.

Note Several functions have similar names but are not part of the user-input group: `acedGetFunCode()`, `acedGetArgs()`, `acedGetVar()`, and `acedGetInput()`.

The following functions behave like user-input functions: `acedEntSel()`, `acedNEntSelP()`, `acedNEntSel()`, and `acedDragGen()`.

The following table briefly describes the user-input functions.

User-input function

summary

Function Name	Description
acedGetInt	Gets an integer value
acedGetReal	Gets a real value
acedGetDist	Gets a distance
acedGetAngle	Gets an angle (oriented to 0 degrees as specified by the ANGBASE variable)
acedGetOrient	Gets an angle (oriented to 0 degrees at the right)
acedGetPoint	Gets a point
acedGetCorner	Gets the corner of a rectangle
acedGetKword	Gets a keyword (see the description of keywords later in this section)
acedGetString	Gets a string

With some user-input functions such as `acedGetString()`, the user enters a value on the AutoCAD prompt line. With others such as `acedGetDist()`, the user either enters a response on the prompt line or specifies the value by selecting points on the graphics screen.

If the screen is used to specify a value, AutoCAD displays rubber-band lines, which are subject to application control. A prior call to `acedInitGet()` can cause AutoCAD to highlight the rubber-band line (or box).

The `acedGetKword()` function retrieves a keyword. Keywords are also string values, but they contain no white space, can be abbreviated, and must be set up before the `acedGetKword()` call by a call to `acedInitGet()`. All user-input functions (except `acedGetString()`) can accept keyword values in addition to the values they normally return, provided `acedInitGet()` has been called to set up the keywords. User-input functions that accept keywords can also accept arbitrary text (with no spaces).

Note You can also use `acedInitGet()` to enable `acedEntSel()`, `acedNEntSelP()`, and `acedNEntSel()` to accept keyword input. The `acedDragGen()` function also recognizes keywords.

The AutoCAD user cannot respond to a user-input function by entering an AutoLISP expression.

The user-input functions take advantage of the error-checking capability of AutoCAD. Trivial errors (such as entering only a single number in response to `acedGetPoint()`) are trapped by AutoCAD and are not returned by the user-input function. The application needs only to check for the conditions shown in the following table.

Return values for user-input functions

Code	Description
RTNORM	User entered a valid value
RTERROR	The function call failed
RTCAN	User entered ESC
RTNONE	User entered only ENTER
RTREJ	AutoCAD rejected the request as invalid
RTKWORD	User entered a keyword or arbitrary text

The `RTCAN` case enables the user to cancel the application's request by pressing ESC. This helps the application conform to the style of built-in AutoCAD commands, which always allow user cancellation. The return values `RTNONE` and `RTKWORD` are governed by the function `acedInitGet()`: a user-input function returns `RTNONE` or `RTKWORD` only if these values have been explicitly enabled by a prior `acedInitGet()` call.

Control of User-Input Function Conditions

The function `acedInitGet()` has two arguments: `val` and `kwl`. The `val` argument specifies one or more control bits that enable or disable certain input values to the following `acedGetxxx()` call. The `kwl` (for keyword list) argument can specify the keywords that the functions `acedGetxxx()`, `acedEntSel()`, `acedNEntSelP()`, `acedNEntSel()`, or `acedDragGen()` recognize.

Note The control bits and keywords established by `acedInitGet()` apply only to the next user-input function call. They are discarded immediately afterward. The application doesn't have to call `acedInitGet()` a second time to clear any special conditions.

Topics in this section

- Input Options for User-Input Functions
- Keyword Specifications

Input Options for User-Input Functions

The following table summarizes the control bits that can be specified by the `val` argument. To set more than one condition at a time, add the values together to create a `val` value between 0 and 127. If `val` is set to zero, none of the control conditions apply to the next user-input function call.

Note Future versions of AutoCAD or ObjectARX may define additional `acedInitGet()` control bits, so you should avoid setting any bits that are not shown in the table or described in this section.

Input options set by acedInitGet()

Code	Bit Value	Description
RSG_NONULL	1	Disallow null input
RSG_NOZERO	2	Disallow zero values
RSG_NONEG	4	Disallow negative values
RSG_NOLIM	8	Do not check drawing limits, even if LIMCHECK is on
RSG_DASH	32	Use dashed lines when drawing rubber-band line or box
RSG_2D	64	Ignore *Z* coordinate of 3D points (acedGetDist() only)
RSG_OTHER	128	Allow arbitrary input—whatever the user enters

The following program excerpt shows the use of `acedInitGet()` to set up a call to the `acedGetInt()` function.

```
int age;
acedInitGet(RSG_NONULL | RSG_NOZERO | RSG_NONEG, NULL);
acedGetInt("How old are you? ", &age);
```

This sequence asks the user's age. AutoCAD automatically displays an error message and repeats the prompt if the user tries to enter a negative or zero value, press ENTER only, or enter a keyword. (AutoCAD itself rejects attempts to enter a value that is not an integer.)

The `RSG_OTHER` option lets the next user-input function call accept arbitrary input. If `RSG_OTHER` is set and the user enters an unrecognized value, the `acedGetxxx()` function returns `RTKWORD`, and the input can be retrieved by a call to `acedGetInput()`. Because spaces end user input just as ENTER does, the arbitrary input never contains a space. The `RSG_OTHER` option has the lowest priority of all the options listed in the preceding table; if the `acedInitGet()` call has disallowed negative numbers with `RSG_NONEG`, for example, AutoCAD still rejects these.

The following code allows arbitrary input (the error checking is minimal).

```
int age, rc;
char userstring[511];
acedInitGet(RSG_NONULL | RSG_NOZERO | RSG_NONEG | RSG_OTHER,
    "Mine Yours");
if ((rc = acedGetInt("How old are you? ", &age))
    == RTKWORD)
// Keyword or arbitrary input
    acedGetInput(userstring);
}
```

In this example, `acedGetInt()` returns the values shown in the following table, depending

on the user's input.

Arbitrary user input	
User Input	**Result**
41	acedGetInt() returns RTNORM and sets age to 41
m	acedGetInt() returns RTKWORD, and acedGetInput() returns "Mine"
y	acedGetInt() returns RTKWORD, and acedGetInput() returns "Yours"
twenty	acedGetInt() returns RTKWORD, and acedGetInput() returns "twenty"
what???	acedGetInt() returns RTKWORD, and acedGetInput() returns "what???"
-10	AutoCAD rejects this input and redisplays the prompt, as RSG_NONEG is set (other bit codes take precedence over RSG_OTHER)
-34.5	acedGetInt() returns RTKWORD, and acedGetInput() returns "-34.5" AutoCAD doesn't reject this value, because it expects an integer, not a real value (if this were an acedGetReal() call, AutoCAD would accept the negative integer as arbitrary input but would reject the negative real value)

Note The `acedDragGen()` function indicates arbitrary input (if this has been enabled by a prior `acedInitGet()` call) by returning `RTSTR` instead of `RTKWORD`.

Keyword Specifications

The optional `kwl` argument specifies a list of keywords that will be recognized by the next user-input (`acedGetxxx()`) function call. The keyword value that the user enters can be retrieved by a subsequent call to `acedGetInput()`. (The keyword value will be available if the user-input function was `acedGetKword()`.) The meanings of the keywords and the action to perform for each is the responsibility of the ObjectARX application.

The `acedGetInput()` function always returns the keyword as it appears in the `kwl` argument, with the same capitalization (but not with the optional characters, if those are specified after a comma). Regardless of how the user enters a keyword, the application has to do only one string comparison to identify it, as demonstrated in the following example. The code segment that follows shows a call to `acedGetReal()` preceded by a call to `acedInitGet()` that specifies two keywords. The application checks for these keywords and sets the input value accordingly.

```
int stat;
ads_real x, pi = 3.14159265;
char kw[20];
// Null input is not allowed.
acedInitGet(RSG_NONULL, "Pi Two-pi");
if ((stat = acedGetReal("Pi/Two-pi/<number>: ", &x)) < 0) {
    if (stat == RTKWORD && acedGetInput(kw) == RTNORM) {
        if (strcmp(kw, "Pi") == 0) {
            x = pi;
            stat = RTNORM;
        } else if (strcmp(kw, "Two-pi") == 0) {
            x = pi * 2;
            stat = RTNORM;
        }
    }
}
if (stat != RTNORM)
    acutPrintf("Error on acedGetReal() input.\n");
else
    acutPrintf("You entered %f\n", x);
```

The call to `acedInitGet()` prevents null input and specifies two keywords: "Pi" and "Two-pi". When `acedGetReal()` is called, the user responds to the prompt Pi/Two-pi/<number> by entering either a real value (stored in the local variable `x`) or one of the keywords. If the user enters a keyword, `acedGetReal()` returns `RTKWORD`. The application retrieves the keyword by calling `acedGetInput()` (note that it checks the error status of this function), and then sets the value of `x` to pi or 2pi, depending on which keyword was entered. In this example, the user can enter either **p** to select pi or **t** to select 2pi.

Graphically Dragging Selection Sets

The function `acedDragGen()` prompts the user to drag a group of selected objects, as shown in the following example:

```
int rc;
ads_name ssname;
ads_point return_pt;
// Prompt the user for a general entity selection.
if (acedSSGet(NULL, NULL, NULL, NULL, ssname) == RTNORM)
// The newly selected entities
    rc = acedDragGen(ssname,
        "Drag selected objects", // Prompt
        0, // Display normal cursor (crosshairs)
        dragsample, // Transformation function
        return_pt); // Set to the specified location.
```

The fourth argument points to a function that does the entity transformation. See Transformation of Selection Setsfor examples of `dragsample()` and `acedDragGen()`.

User Breaks

The user-input functions and the acedCommand(), acedCmd(), acedEntSel(), acedNEntSelP(), acedNEntSel(), acedDragGen(), and acedSSGet() functions return RTCAN if the AutoCAD user responds by pressing ESC. An external function should treat this response as a cancel request and return immediately. ObjectARX also provides a function, acedUsrBrk(), that explicitly checks whether the user pressed ESC. This function enables ObjectARX applications to check for a user interrupt.

An application doesn't need to call acedUsrBrk() unless it performs lengthy computation between interactions with the user. The function acedUsrBrk() should never be used as a substitute for checking the value returned by user-input functions that can return RTCAN.

In some cases, an application will want to ignore the user's cancellation request. If this is the case, it should call acedUsrBrk() to clear the request; otherwise, the ESC will still be outstanding and will cause the next user-input call to fail. (If an application ignores the ESC, it should print a message to tell the user it is doing so.) Whenever an ObjectARX application is invoked, the ESC condition is automatically cleared.

For example, the following code fragment fails if the user enters ESC at the prompt.

```
int test()
{
    int i;
    while (!acedUsrBrk()) {
        acedGetInt("\nInput integer:", &i); // WRONG
        .
        .
        .
    }
}
```

The slightly modified code fragment that follows correctly handles an input of ESC without calling acedUsrBrk().

```
int test()
{
    int i;
    for (;;) {
        if (acedGetInt("\nInput integer:", &i) != RTNORM)
            break;
        ...
    }
}
```

The following sample changes the loop condition. This construction also works correctly.

```
int test()
{
    int i;
    while (acedGetInt("\nInput integer:", &i) == RTNORM) {
        ...
    }
}
```

A valid place to use acedUsrBrk() is in a lengthy operation. For example, code that steps through every entity in the drawing database can be time consuming and should call acedUsrBrk().

Returning Values to AutoLISP Functions

ObjectARX provides a set of functions that enables an external function to return values to AutoLISP. These value-returnfunctions have no AutoLISP counterparts. The following table summarizes these functions.

Value-return function summary

Function Name	Returns
acedRetInt	An integer value
acedRetReal	A real value
acedRetPoint	A 3D point
acedRetStr	A string
acedRetVal	A value passed "generically" in a result buffer
acedRetName	An entity (RTENAME) or selection set (RTPICKS) name (see Selection Set and Entity Namesfor more information on selection sets and entities)
acedRetT	The AutoLISP value t (true)
acedRetNil	The AutoLISP value nil
acedRetVoid	A blank value: AutoCAD doesn't display the result
acedRetList	A list of result buffers returned to AutoLISP

The following example shows the scheme of a function called when the application receives a `kInvkSubrMsg` request. It returns a real value to AutoLISP.

```
int dofun()
{
    ads_real x
// Check the arguments and input conditions here.
// Calculate the value of x.
    acedRetReal(x);
    return GOOD;
}
```

Note An external function can make more than one call to value-return functions upon a single `kInvkSubrMsg` request, but the AutoLISP function returns only the value passed it by the last value-return function invoked.

Conversions

The functions described in this section are utilities for converting data types and units.

Topics in this section

- String Conversions
- Real-World Units

String Conversions

The functions `acdbRToS()` and `acdbAngToS()` convert values used in AutoCAD to string values that can be used in output or as textual data. The `acdbRToS()` function converts a real value, and `acdbAngToS()` converts an angle. The format of the result string is controlled by the value of AutoCAD system variables: the units and precision are specified by LUNITS and LUPREC for real (linear) values and by AUNITS and AUPREC for angular values. For both functions, the DIMZIN dimensioning variable controls how leading and trailing zeros are written to the result string. The complementary functions `acdbDisToF()` and `acdbAngToF()` convert strings back into real (distance) values or angles. If passed a string generated by `acdbRToS()` or `acdbAngToS()`, `acdbDisToF()` and `acdbAngToF()` (respectively) are guaranteed to return a valid value.

For example, the following fragment shows calls to `acdbRToS()`. (Error checking is not shown but should be included in applications.)

```
ads_real x = 17.5;
char fmtval[12];
//Precision is the 3rd argument: 4 places in the first
// call, 2 places in the others.
acdbRToS(x, 1, 4, fmtval); // Mode 1 = scientific
acutPrintf("Value formatted as %s\n", fmtval);
acdbRToS(x, 2, 2, fmtval); // Mode 2 = decimal
acutPrintf("Value formatted as %s\n", fmtval);
acdbRToS(x, 3, 2, fmtval); // Mode 3 = engineering
acutPrintf("Value formatted as %s\n", fmtval);
acdbRToS(x, 4, 2, fmtval); // Mode 4 = architectural
acutPrintf("Value formatted as %s\n", fmtval);
acdbRToS(x, 5, 2, fmtval); // Mode 5 = fractional
acutPrintf("Value formatted as %s\n", fmtval);
```

These calls (assuming that the DIMZIN variable equals 0) display the following values on the AutoCAD text screen.

Value formatted as 1.7500E+01

Value formatted as 17.50

Value formatted as 1’-5.50”

Value formatted as 1’-5 1/2”

Value formatted as 17 1/2

When the UNITMODE system variable is set to 1, which specifies that units are displayed

as entered, the string returned by acdbRToS() differs for engineering (mode equals 3), architectural (mode equals 4), and fractional (mode equals 5) units. For example, the first two lines of the preceding sample output would be the same, but the last three lines would appear as follows:

Value formatted as 1’5.50”

Value formatted as 1’5-1/2”

Value formatted as 17-1/2

The acdbDisToF() function complements acdbRToS(), so the following calls, which use the strings generated in the previous examples, all set result to the same value, 17.5. (Again, the examples do not show error checking.)

```
acdbDisToF("1.7500E+01", 1, &result); // 1 = scientific
acdbDisToF("17.50", 2, &result); // 2 = decimal
// Note the backslashes. Needed for inches.
acdbDisToF("1'-5.50\"", 3, &result); // 3 = engineering
acdbDisToF("1'-5 1/2\"", 4, &result); // 4 = architectural
acdbDisToF("17 1/2", 5, &result); // 5 = fractional
```

The following fragment shows calls to acdbAngToS() that are similar to the previous acdbRToS() examples.

```
ads_real ang = 3.14159;
char fmtval[12];
// Precision is the 3rd argument: 0 places in the first
// call, 4 places in the next 3, 2 in the last.
acdbAngToS(ang, 0, 0, fmtval); // Mode 0 = degrees
acutPrintf("Angle formatted as %s\n", fmtval);
acdbAngToS(ang, 1, 4, fmtval); // Mode 1 = deg/min/sec
acutPrintf("Angle formatted as %s\n", fmtval);
acdbAngToS(ang, 2, 4, fmtval); // Mode 2 = grads
acutPrintf("Angle formatted as %s\n", fmtval);
acdbAngToS(ang, 3, 4, fmtval); // Mode 3 = radians
acutPrintf("Angle formatted as %s\n", fmtval);
acdbAngToS(ang, 4, 2, fmtval); // Mode 4 = surveyor's
acutPrintf("Angle formatted as %s\n", fmtval);
```

These calls (still assuming that DIMZIN equals 0) display the following values on the AutoCAD text screen.

Angle formatted as 180

Angle formatted as 180d0’0”

Angle formatted as 200.0000g

Angle formatted as 3.1416r

Angle formatted as W

Note The UNITMODE system variable also affects strings returned by acdbAngToS() when it returns a string in surveyor's units (mode equals 4). If UNITMODE equals 0, the string returned can include spaces (for example, “N 45d E”); if UNITMODE equals 1, the string contains no spaces (for example, “N45dE”).

The acdbAngToF() function complements acdbAngToS(), so the following calls all set the result argument to the same value, 3.14159. (This is rounded up to 3.1416 in the example that uses radians.)

```
acdbAngToF("180", 0, &result); // 0 = degrees
acdbAngToF("180d0'0\"", 1, &result); // 1 = deg/min/sec
acdbAngToF("200.0000g", 2, &result); // 2 = grads
acdbAngToF("3.1416r", 3, &result); // 3 = radians
acdbAngToF("W", 4, &result); // 4 = surveyor's
```

Note When you have a string that specifies an angle in degrees, minutes, and seconds, you must use a backslash (\) to escape the seconds symbol (") so that it doesn't appear to be the end of the string. The second of the preceding acdbAngToF() examples demonstrates this.

Real-World Units

The file *acad.unt* defines a variety of conversions between real-world units such as miles/kilometers, Fahrenheit/Celsius, and so on. The function acutCvUnit() takes a value expressed in one system of units and returns the equivalent value in another system. The two systems of units are specified by strings that must match one of the definitions in *acad.unt*.

If the current drawing units are engineering or architectural (feet and inches), the following fragment converts a user-specified distance into meters.

```
ads_real eng_len, metric_len;
char *prmpt = "Select a distance: ";
if (acedGetDist(NULL, prmpt, &eng_len) != RTNORM)
    return BAD;
acutCvUnit(eng_len, "inches", "meters", &metric_len);
```

The acutCvUnit() function will not convert incompatible units, such as inches into years.

Character Type Handling

ObjectARX provides a package of character-handling functions, as shown in the table that follows. The advantage of this package over the standard C library package, *ctype.h*, is that these functions are independent of any specific character set and are not bound to ASCII. They are customized to the current AutoCAD language configuration. In other respects, they behave like their standard C counterparts.

Character type functions

Function Name	Purpose
acutIsAlpha	Verifies that the character is alphabetic

acutIsUpper	Verifies that the character is uppercase
acutIsLower	Verifies that the character is lowercase
acutIsDigit	Verifies that the character is a digit
acutIsXDigit	Verifies that the character is a hexadecimal digit
acutIsSpace	Verifies that the character is a white-space character
acutIsPunct	Verifies that the character is a punctuation character
acutIsAlNum	Verifies that the character is alphanumeric
acutIsPrint	Verifies that the character is printable
acutIsGraph	Verifies that the character is graphical
acutIsCntrl	Verifies that the character is a control character
acutToUpper	Converts the character to uppercase
acutToLower	Converts the character to lowercase

The following code fragment takes a character (the value in this example is arbitrary) and converts it to uppercase. The `acutToUpper()` function has no effect if the character is already uppercase.

```
int cc = 0x24;
cc = acutToUpper(cc);
```

Coordinate System Transformations

The `acedTrans()` function translates a point or a displacement from one coordinate system into another. It takes a point argument, `pt`, that can be interpreted as either a three-dimensional point or a three-dimensional displacement vector. This is controlled by an argument called `disp`, which must be nonzero if `pt` is treated as a displacement vector; otherwise, `pt` is treated as a point. The translated point or vector is returned in a call-by-reference result argument, which, like `pt`, is of type `ads_point`.

The arguments that specify the two coordinate systems, `from` and `to`, are both result buffers. The `from` argument specifies the coordinate system in which `pt` is expressed, and the `to` argument specifies the coordinate system of the result. Both the `from` and `to` arguments can specify a coordinate system in any of the following ways:

- An integer code (`restype == RTSHORT`) that specifies the WCS, current UCS, or current DCS (of either the current viewport or paper space).
- An entity name (`restype == RTENAME`), as returned by one of the entity name or selection set functions. This specifies the ECS of the named entity. For planar entities,

the ECS can differ from the WCS. If the ECS does not differ, conversion between ECS and WCS is an identity operation.

- A 3D extrusion vector (`restype == RT3DPOINT`), which is another method of specifying an entity's ECS. Extrusion vectors are always represented in world coordinates; an extrusion vector of (0,0,1) specifies the WCS itself.

The following are descriptions of the AutoCAD coordinate systems that can be specified by the `from` and `to` arguments.

WCS

World Coordinate System. The "reference" coordinate system. All other coordinate systems are defined relative to the WCS, which never changes. Values measured relative to the WCS are stable across changes to other coordinate systems.

UCS

User Coordinate System. The "working" coordinate system. All points passed to AutoCAD commands, including those returned from AutoLISP routines and external functions, are points in the current UCS (unless the user precedes them with a * at the Command prompt). If you want your application to send coordinates in the WCS, ECS, or DCS to AutoCAD commands, you must first convert them to the UCS by calling `acedTrans()`.

ECS

Entity Coordinate System. Point values returned by `acdbEntGet()` are expressed in this coordinate system relative to the entity itself. Such points are useless until they are converted into the WCS, current UCS, or current DCS, according to the intended use of the entity. Conversely, points must be translated into an ECS before they are written to the database by means of `acdbEntMod()` or `acdbEntMake()`.

DCS

Display Coordinate System. The coordinate system into which objects are transformed before they are displayed. The origin of the DCS is the point stored in the AutoCAD TARGET system variable, and its *Z* axis is the viewing direction. In other words, a viewport is always a plan view of its DCS. These coordinates can be used to determine where something appears to the AutoCAD user.

When the `from` and `to` integer codes are 2 and 3, in either order, 2 indicates the DCS for the current model space viewport, and 3 indicates the DCS for paper space (PSDCS). When the 2 code is used with an integer code other than 3 (or another means of specifying the coordinate system), it is assumed to indicate the DCS of the current space (paper space or model space), and the other argument is assumed to indicate a coordinate system in the current space.

PSDCS

Paper Space DCS. This coordinate system can be transformed only to or from the DCS of the currently active model space viewport. This is essentially a 2D transformation, where the *X* and *Y* coordinates are always scaled and are offset if the `disp` argument is 0. The *Z* coordinate is scaled but is never translated; it can be used to find the scale factor between the two coordinate systems. The PSDCS (integer code 2) can be transformed only into the current model space viewport: if the `from` argument equals 3, the `to` argument must equal 2, and vice versa.

The following example translates a point from the WCS into the current UCS.

```
ads_point pt, result;
struct resbuf fromrb, torb;
pt[X] = 1.0;
pt[Y] = 2.0;
pt[Z] = 3.0;
fromrb.restype = RTSHORT;
fromrb.resval.rint = 0; // WCS
torb.restype = RTSHORT;
torb.resval.rint = 1; // UCS
// disp == 0 indicates that pt is a point:
acedTrans(pt, &fromrb, &torb, FALSE, result);
```

If the current UCS is rotated 90 degrees counterclockwise around the world *Z* axis, the call to `acedTrans()` sets the result to the point (2.0,-1.0,3.0). However, if `acedTrans()` is called as shown in the following example, the result is (-2.0,1.0,3.0).

```
acedTrans(pt, &torb, &fromrb, FALSE, result);
```

Display Control

ObjectARX has several functions for controlling the AutoCAD display, including both text and graphics screens.

Topics in this section

- Interactive Output
- Control of Graphics and Text Screens
- Control of Low-Level Graphics and User Input

Interactive Output

The basic output functions are `acedPrompt()`, which displays a message on the AutoCAD prompt line, and `acutPrintf()`, which displays text on the text screen. The `acutPrintf()` function's calling sequence is equivalent to the standard C library function `printf()`. It is provided as a separate function, because on some platforms the standard C `printf()` causes the output message to mangle the AutoCAD graphics screen. (Remember that the `acdbFail()` function also displays messages on the text screen.)

The size of a string displayed by `acedPrompt()` should not exceed the length of the graphics screen's prompt line; typically this is no more than 80 characters. The size of a string displayed by `acutPrintf()` must not exceed 132 characters, because this is the size of the string buffer used by the `acutPrintf()` function (133 bytes, with the last byte reserved for the null character).

The `acedMenuCmd()` function provides control of the display of the graphics screen menu. The `acedMenuCmd()` function activates one of the submenus of the current menu. It takes a

string argument, `str`, that consists of two parts, separated by an equal sign, in the form:

```
"section=submenu"
```

where `section` indicates the menu section and `submenu` indicates which submenu to activate within that section.

For example, the following function call causes the OSNAP submenu defined in the current customization file to appear on the screen.

```
acedMenuCmd("S=OSNAP");
```

In a similar way, the following function call assigns the submenu MY–BUTTONS to the BUTTONS menu, and activates it.

```
acedMenuCmd("B=MY-BUTTONS");
```

In Release 12 and earlier versions of AutoCAD, you could assign any kind of menu to any other. For example, you could assign a SCREEN menu to a POP menu. With Release 13 and later versions of AutoCAD, you can assign menus to other menus on the Windows platform only if they are of the same type. A POP menu can be assigned only to another POP menu, and a SCREEN menu to another SCREEN menu. You can specify the menu in detail, because Windows loads partial menus.

Calling `acedMenuCmd()` and passing "P1=test.numeric" assigns POP menu 12 to POP menu 2, assuming that a customization group named "test" is currently loaded and it has a menu with the aliases "POP12" and "numeric" defined.

The following call shows how to activate a drop-down menu and then display it.

```
acedMenuCmd("P1=NUMERIC");
```

The call to `acedMenuCmd()` assigns the submenu NUMERIC to drop-down menu 1 (in the upper-left corner of the graphics screen).

See the *AutoCAD Customization Guide* for more information on custom menus.

Control of Graphics and Text Screens

On single-screen AutoCAD installations, an ObjectARX application can call `acedGraphScr()` to display the graphics screen or `acedTextScr()` to display the text screen. These functions are equivalent to the AutoCAD GRAPHSCR and TEXTSCR commands or to toggling the Flip Screen function key.

The `acedRedraw()` function is similar to the AutoCAD REDRAW command, but it provides more control over what is displayed: it can redraw the entire graphics screen and also specify a single object to be either redrawn or undrawn (blanked out). If the object is a complex object such as a polyline or block, `acedRedraw()` can draw (or undraw) either the entire object or only its header. The `acedRedraw()` function can be used also to highlight or unhighlight selected objects.

Control of Low-Level Graphics and User Input

Certain functions provide direct access to the AutoCAD graphics screen and input devices. They enable ObjectARX applications to use some of the display and user-interaction facilities built into AutoCAD.

The `acedGrText()` function displays text in the status or menu areas, with or without highlighting. The `acedGrDraw()` function draws a vector in the current viewport, with control over color and highlighting. The `acedGrVecs()` function draws multiple vectors. The `acedGrRead()` function returns "raw" user input, whether from the keyboard or the pointing device; if the call to `acedGrRead()` enables tracking, the function returns digitized coordinates that can be used for dragging.

Warning Because these functions depend on code in AutoCAD, their operation can change from release to release. Applications that call these functions may not be upward compatible. Also, they depend on the current hardware configuration. In particular, applications that call `acedGrText()` and `acedGrRead()` are not likely to work the same on all configurations unless the developer uses them as described earlier to avoid hardware-specific features. These functions do almost no error reporting and can damage the graphics screen display (see the example for a way to fix this problem).

The following sequence reverses damage to the graphics screen display caused by incorrect calls to `acedGrText()`, `acedGrDraw()`, or `acedGrVecs()`.

```
acedGrText(-3, NULL, 0);
acedRedraw(NULL, 0);
```

The arguments to `acedGrText()` have the following meanings: -3 restores standard text, `NULL` == no new text, and 0 == no highlighting. The arguments to `acedRedraw()` have the following meanings: `NULL` == all entities, and 0 == entire viewport.

Tablet Calibration

AutoCAD users with a digitizing tablet can calibrate the tablet by using the TABLET command. With the `acedTablet()` function, applications can manage calibrations by setting them directly and by saving calibration settings for future use. The function takes two arguments, `list` and `result`, each of which is a result-buffer list. The first result buffer in the first list is an integer code that must be 0 to retrieve the current calibration (in `result`), or 1 to set the calibration according to the remaining buffers in list. Calibrations are expressed as four 3D points (in addition to the code). The first three of these points—row1, row2, and row3—are the three rows of the tablet's transformation matrix. The fourth point is a vector, direction, that is normal to the plane of the tablet's surface (expressed in WCS).

Note The TABMODE system variable controls whether Tablet mode is set to On (1) or Off (0). You can control it by using `acedSetVar()`.

The following code sequence retrieves the current tablet calibration, and saves it in `calibr2`. In this example, the user has used the TABLET command to calibrate the matrix,

and Tablet mode is on.

```
struct resbuf *calibr1, *calibr2;
struct resbuf varbuf, rb;
// Retrieve the current calibration.
calibr1 = acutBuildList(RTSHORT, 0, RTNONE);
if (acedTablet(calibr1, &calibr2) != RTNORM) {
    acdbFail("Calibration not obtainable\n");
    return BAD;
}
```

The code returned in the result argument, `calibr2` in the example, is automatically set to 1. To reset the calibration to the values retrieved by the preceding example, you could use the following code:

```
if (acedTablet(calibr2, &calibr1) != RTNORM) {
    acdbFail("Couldn't reset calibration\n");
    return BAD;
}
rb.restype = RTSHORT;
rb.resval.rint = 1;
acedSetVar("TABMODE", &rb);
acedGetVar("TABMODE" &varbuf);
if (varbuf.resval.rint == 0) {
    acdbFail("Couldn't set TABMODE\n");
    return BAD;
}
```

In this example, `calibr1` now contains the result of the calibration. Because this is presumably identical to `calibr2` (which was initialized by `acedTablet()`), you don't necessarily need this result. When you set a calibration, you can specify a `NULL` result, which causes `acedTablet()` to set the calibration "silently."

```
if (acedTablet(calibr2, NULL) != RTNORM) { . . . }
```

The transformation matrix passed as row1, row2, and row3 is a 3x3 transformation matrix meant to transform a 2D point. The 2D point is expressed as a column vector in homogeneous coordinates (by appending 1.0 as the third element), so the transformation looks like this:

$$\begin{bmatrix} X' \\ Y' \\ D' \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & 1.0 \end{bmatrix} \bullet \begin{bmatrix} X' \\ Y' \\ 1.0 \end{bmatrix}$$

The calculation of a point is similar to the 3D case. AutoCAD transforms the point by using the following formulas:

$$X' = M_{00}X + M_{01}Y + M_{02}$$

$$Y' = M_{10}X + M_{11}Y + M_{12}$$

$$D' = M_{20}X + M_{21}Y + 1.0$$

To turn the resulting vector back into a 2D point, the first two components are divided by the third, the scale factor D', yielding the point $(X'/D', Y'/D')$.

For a projective transformation, which is the most general case, `acedTablet()` does the full calculation. But for affine and orthogonal transformations, M_{20} and M_{21} are both 0, so D' would be 1.0. The calculation of D' and the division are omitted; the resulting 2D point is simply (X', Y').

An affine transformation is a special, uniform case of a projective trans-formation. An orthogonal transformation is a special case of an affine transformation: not only are M_{20} and M_{21} 0, but $M_{00} = M_{11}$ and $M_{10} = -M_{01}$.

Note When you set a calibration, the result does not equal the list argument if the direction in the list was not normalized; AutoCAD normalizes the direction vector before it returns it. Also, it ensures that the third element in the third column (row3[Z]) is equal to 1. This situation should not arise if you set the calibration using values retrieved from AutoCAD by means of `acedTablet()`. However, it can happen if your program calculates the transformation itself.

Wild-Card Matching

The `acutWcMatch()` function enables applications to compare a string to a wild-card pattern. This facility can be used when building a selection set (in conjunction with `acedSSGet()`) and when retrieving extended entity data by application name (in conjunction with `acdbEntGetX()`).

The `acutWcMatch()` function compares a single string to a pattern, and returns `RTNORM` if the string matches the pattern, and `RTERROR` if it does not. The wild-card patterns are similar to the regular expressions used by many system and application programs. In the pattern, alphabetic characters and numerals are treated literally; brackets can be used to specify optional characters or a range of letters or digits; a question mark (?) matches a single character, and an asterisk (*) matches a sequence of characters; certain other special characters have meanings within the pattern. For a complete table of characters used in wild-card strings, see the description of `acutWcMatch()`.

In the following examples, a string variable called `matchme` has been declared and initialized. The following call checks whether `matchme` begins with the five characters "allof".

```
if (acutWcMatch(matchme, "allof*") == RTNORM) {
    .
    .
    .
}
```

The following call illustrates the use of brackets in the pattern. In this case, `acutWcMatch()` returns `RTNORM` if `matchme` equals "STR1", "STR2", "STR3", or "STR8".

```
if (acutWcMatch(matchme, "STR[1-38]") == RTNORM) {
    .
    .
    .
}
```

The pattern string can specify multiple patterns, separated by commas. The following call returns `RTNORM` if `matchme` equals "ABC", if it begins with "XYZ", or if it ends with "123".

```
if (acutWcMatch(matchme, "ABC,XYZ*,*123") == RTNORM) {
    .
    .
    .
}
```

The `acutWcMatchEx()` function is similar to `acutWcMatch()`, but it has an additional argument to allow it to ignore case.

```
bool
acutWcMatchEx(
    const char * string,
    const char * pattern,
    bool ignoreCase);
```

ObjectARX Global Utility Functions

This section discusses some general characteristics of the ObjectARX ® global utility functions. For more information on specific functions, see the *ObjectARX Reference*.

Topics in this section

- Common Characteristics of ObjectARX Library Functions
- Variables, Types, and Values Defined in ObjectARX
- Lists and Other Dynamically Allocated Data
- Extended Data Exclusive Data Types
- Text String Globalization Issues

Common Characteristics of ObjectARX Library Functions

This section describes some general characteristics of global functions in the ObjectARX library. Most ObjectARX global functions that operate on the database, system variables, and selection sets work on the current document.

Note The functions described in this section were known as the ADS functions in previous releases of AutoCAD ® .

Topics in this section

- ObjectARX Global Function Calls Compared to AutoLISP Calls
- Function Return Values versus Function Results
- External Functions
- Error Handling Functions
- Communication Between Applications
- Handling External Applications

ObjectARX Global Function Calls Compared to AutoLISP Calls

Many ObjectARX global functions are unique to the ObjectARX programming environment, but many provide the same functionality as AutoLISP ® functions: they have the same name as the comparable AutoLISP function, except for the prefix (aced, acut, etc.). This similarity makes it easy to convert programs from AutoLISP to ObjectARX. However, there are important differences between the interpretive AutoLISP environment and the compiled C++ environment.

Topics in this section

- Argument Lists in AutoLISP and C
- Memory Management

Argument Lists in AutoLISP and C

Many built-in AutoLISP functions accept an arbitrary number of arguments. This is natural for the LISP environment, but to require variable-length argument lists for every comparable function in the ObjectARX library would impose needless complexity. To avoid this problem, a simple rule was applied to the library: an ObjectARX function that is an analog of an AutoLISP function takes all arguments that the AutoLISP function takes. Where an argument is optional in AutoLISP, in the ObjectARX library a special value, usually a null pointer, 0, or 1, can be passed to indicate that the option is not wanted.

A few ObjectARX library functions are exceptions to this rule. The `acutPrintf()` function is similar to the standard C library `printf()` function. Like the standard version, it is implemented as a variadic function; that is, it takes a variable-length argument list. The AutoLISP `command` function not only accepts a variable number of arguments of various types, but it also accepts types defined especially for AutoCAD, such as points and selection sets. In addition, the AutoLISP `entget` function has an optional argument for retrieving extended data. In ObjectARX, the `acdbEntGet()` function does not have a corresponding argument. Instead, there is an additional function, `acdbEntGetX()`, provided specifically for retrieving extended data.

Memory Management

The memory requirements of an ObjectARX application are different from those of AutoLISP. On the one hand, the data structures employed by C++ programs tend to be more compact than AutoLISP lists. On the other hand, there is a rather large, fixed overhead for running ObjectARX applications. Part of this consists of code that must be present in the applications themselves; the larger part is the ObjectARX library.

Some ObjectARX global functions allocate memory automatically. In most cases, the application must explicitly release this memory as if the application itself had allocated it. AutoLISP has automatic garbage collection, but ObjectARX does not.

Warning

Failure to release automatically allocated memory slows down the system and can cause AutoCAD to terminate.

Function Return Values versus Function Results

Many ObjectARX global functions return an integer status code that indicates whether the function call succeeded or failed.

The code `RTNORM` indicates that the function succeeded; other codes indicate failure or special conditions. Library functions that return a status code pass their actual results (if any) back to the caller through an argument that is passed by reference. To determine how a particular global function uses its arguments and return values, consult its reference documentation.

Consider the following prototyped declarations for a few typical ObjectARX functions:

```
int acdbEntNext(ads_name ent, ads_name result);
int acedOsnap(ads_point pt, char *mode, ads_point
    result);
int acedGetInt(char *prompt, int *result);
```

An application could call these functions with the following C++ statements:

```
stat = acdbEntNext(ent, entres);
stat = acedOsnap(pt, mode, ptres);
stat = acedGetInt(prompt, &intres);
```

After each function is called, the value of the stat variable indicates either success (`stat == RTNORM`) or failure (`stat == RTERROR` or another error code, such as `RTCAN` for cancel). The last argument in each list is the result argument, which must be passed by reference. If successful, `acdbEntNext()` returns an entity name in its `entres` argument, `acedOsnap()` returns a point in `ptres`, and `acedGetInt()` returns an integer result in `intres`. (The types `ads_name` and `ads_point` are array types, which is why the `entres` and `ptres` arguments don't explicitly appear as pointers.)

External Functions

Once an ObjectARX application has defined its external functions (with calls to `acedDefun`

()), the functions can be called by the AutoLISP user and by AutoLISP programs and functions as if they were built-in or user-defined AutoLISP functions. An external function can be passed AutoLISP values and variables, and can return a value to the AutoLISP expression that calls it. Some restrictions apply and are described in this section.

Topics in this section

- Defining External Functions
- Evaluating External Functions

Defining External Functions

When an ObjectARX application receives a `kLoadDwgMsg` request from AutoCAD, it must define all of its external functions by calling `acedDefun()` once for each function. The `acedDefun()` call associates the external function's name (passed as a string value) with an integer code that is unique within the application. The integer code must not be negative, and it cannot be greater than 32,767 (in other words, the code is a short integer).

The following call to `acedDefun()` specifies that AutoLISP will recognize an external function called `doit` in AutoLISP, and that when AutoLISP invokes `doit`, it passes the function code zero (0) to the ObjectARX application:

```
acedDefun("doit", 0);
```

The string that specifies the name of the new external function can be any valid AutoLISP symbol name. AutoLISP converts it to all uppercase and saves it as a symbol of the type `Exsubr`.

External functions are defined separately for each open document in the MDI. The function gets defined when the document becomes active. For more information, see The Multiple Document Interface.

Warning If two or more ObjectARX applications define functions (in the same document) that have the same name, AutoLISP recognizes only the most recently defined external function. The previously loaded function will be lost. This can also happen if the user calls `defun` with a conflicting name.

As in AutoLISP, the new function can be defined as an AutoCAD command by prefixing its name with "C:" or "c:", as shown in the following example:

```
acedDefun("C:DOIT", 0);
```

In this case, DOIT can now be invoked from the AutoCAD Command prompt without enclosing its name in parentheses.

Functions defined as AutoCAD commands can still be called from AutoLISP expressions, provided that the "C:" prefix is included as a part of their names. For example, given the previous `acedDefun()` call, the AutoCAD user could also invoke the DOIT command as a function with arguments:

Command: **(c:doit x y)**

Warning If the application defines a `C:XXX` command whose name conflicts with a built-in command or a command name defined in the acad.pgp file, AutoCAD does not recognize the external function as a command. The function can still be invoked as an AutoLISP external function. For example, after the call `acedDefun("c:cp", 0)`, a user input of `cp` (an alias for COPY defined in acad.pgp) invokes the AutoCAD COPY command, but the user could invoke the external function with `c:cp`.

Note Function names defined by `acedDefun()` can be undefined by calling `acedUndef()`. After a function has been undefined, an attempt to invoke it causes an error.

Evaluating External Functions

Once an external function has been defined, AutoLISP can invoke it with an `kInvkSubrMsg` request. When the ObjectARX application receives this request, it retrieves the external function's integer code by calling `acedGetFunCode()`. Then a switch statement, an if statement, or a function–call table can select and call the indicated function handler. This is the function that the ObjectARX application defines to implement the external function. Note that the name of the handler and the name defined by `acedDefun()` (and therefore recognized by AutoLISP) are not necessarily the same name.

If the function handler expects arguments, it can retrieve their values by calling `acedGetArgs()`, which returns a pointer to a linked list of result buffers that contain the values passed from AutoLISP. If the handler expects no arguments, it does not need to call `acedGetArgs()` (it can do so anyway, to verify that no arguments were passed). Because it retrieves its arguments from a linked list, the function handler can also implement variable-length argument lists or varying argument types.

Note

The function handler must verify the number and type of arguments passed to it, because there is no way to tell AutoLISP what the requirements are.

Function handlers that expect arguments can be written so that they prompt the user for values if `acedGetArgs()` returns a `NULL` argument list. This technique is often applied to external functions defined as AutoCAD commands.

A group of ObjectARX functions known as value-return functions (such as `acedRetInt()`, `acedRetReal()`, and `acedRetPoint()`) enable an external function to return a value to the AutoLISP expression that invoked it.

Arguments that are passed between external functions and AutoLISP must evaluate to one of the following types: integer, real (floating-point), string, point (represented in AutoLISP as a list of two or three real values), an entity name, a selection set name, the AutoLISP symbols `t` and `nil`, or a list that contains the previous elements. AutoLISP symbols other than `t` and `nil` are not passed to or from external functions, but an ObjectARX application can retrieve and set the value of AutoLISP symbols by calling `acedGetSym()` and `acedPutSym()`.

If, for example, an external function in an ObjectARX application is called with a string, an integer, and a real argument, the AutoLISP version of such a function can be represented as follows:

```
(doitagain pstr iarg rarg)
```

Assuming that the function has been defined with `acedDefun()`, an AutoCAD user can invoke it with the following expression:

Command: (doitagain "Starting width is" 3 7.12)

This call supplies values for the function's string, integer, and real number arguments, which the `doitagain()` function handler retrieves by a call to `acedGetArgs()`. For an example of retrieving arguments in this way, see the first example in Lists and Other Dynamically Allocated Data.

Error Handling Functions

The AutoCAD environment is complex and interactive, so ObjectARX applications must be robust. ObjectARX provides several error-handling facilities. The result codes returned during "handshaking" with AutoLISP indicate error conditions, as do the result codes library functions returned to the application. Functions that prompt for input from the AutoCAD user employ the built-in input-checking capabilities of AutoCAD. In addition, three functions let an application notify users of an error: `acdbFail()`, `acedAlert()`, and `acrx_abort()`.

The `acdbFail()` function simply displays an error message (passed as a single string) at the AutoCAD Command prompt. This function can be called to identify recoverable errors such as incorrect argument values passed by the user.

The statement in the following example calls `acdbFail()` from a program named *test.arx*:

```
acdbFail("invalid osnap point\n");
```

The `acdbFail()` function displays the following:

Application test.arx ERROR: invalid osnap point

You can also warn the user about error conditions by displaying an alert box. To display an alert box, call `acedAlert()`. Alert boxes are a more emphatic way of warning the user, because the user has to choose OK before continuing.

For fatal errors, `acrx_abort()` should be called. This function prompts the user to save work in progress before exiting. The standard C++ `exit()` function should not be called.

To obtain detailed information about the failure of an ObjectARX function, inspect the AutoCAD system variable ERRNO. When certain ObjectARX function calls (or AutoLISP function calls) cause an error, ERRNO is set to a value that the application can retrieve by a call to `acedGetVar()`. ObjectARX defines symbolic names for the error codes in the header file *ol_errno.h*, which can be included by ObjectARX applications that examine ERRNO. These codes are shown in the *ObjectARX Reference*.

Communication Between Applications

The ObjectARX function acedInvoke() in one application is used to call external functions defined and implemented by other ObjectARX applications. The external function called by acedInvoke() must be defined by a currently loaded ObjectARX application.

The acedInvoke() function calls the external function by the name that its application has specified in the acedDefun() call, which is the function name that AutoLISP calls to invoke the function. If the external function was defined as an AutoLISP command, with "C:" as a prefix to its name, these characters must be included in the string that acedInvoke() specifies (as when the command is invoked with an AutoLISP expression).

Warning Because applications loaded at the same time cannot have duplicate function names, you should take this into account when designing an application that uses more than a single program file; avoid the problem with a naming scheme or convention that ensures that the name of each external function will be unique. The best solution is to use your Registered Developer Symbol (RDS) as a prefix.

The name of the external function, and any argument values that it requires, is passed to acedInvoke() in the form of a linked list of result buffers. It also returns its result in a result-buffer list; the second argument to acedInvoke() is the address of a result-buffer pointer.

The following sample function calls acedInvoke() to invoke the factorial function fact() in the sample program fact.cpp:

```
static void test()
{
    int stat, x = 10;
    struct resbuf *result = NULL, *list;
    // Get the factorial of x from file fact.cpp.
    list = acutBuildList(RTSTR, "fact", RTSHORT, x, RTNONE);
    if (list != NULL) {
        stat = acedInvoke(list, &result);
        acutRelRb(list);
    }
    if (result != NULL) {
        acutPrintf("\nSuccess: factorial of %d is %d\n", x,
            result->resval.rint);
        acutRelRb(result);
    }
    else
        acutPrintf("Test failed\n");
}
```

If a function is meant to be called with acedInvoke(), the application that defines it should register the function by calling acedRegFunc(). (In some cases the acedRegFunc() call is required, as described later in this section.) When acedRegFunc() is called to register the function, ObjectARX calls the function directly, without going through the application's dispatch loop. To define the function, call acedRegFunc().

An external function handler registered by acedRegFunc() must have no arguments and must return an integer (which is one of the application result codes—either RSRSLT or RSERR).

The following excerpt shows how the funcload() function in fact.cpp can be modified to register its functions as well as define them:

```
typedef int (*ADSFUNC) (void);
// First, define the structure of the table: a string
// giving the AutoCAD name of the function, and a pointer to
// a function returning type int.
struct func_entry { char *func_name; ADSFUNC func; };
// Declare the functions that handle the calls.
int fact (void); // Remove the arguments
int squareroot (void);
// Here we define the array of function names and handlers.
//
static struct func_entry func_table[] =
    {  {"fact", fact},
    {"sqr", squareroot},
    };
...
static int funcload()
{
    int i;
    for (i = 0; i < ELEMENTS(func_table); i++) {
        if (!acedDefun(func_table[i].func_name, i))
            return RTERROR;
        if (!acedRegFunc(func_table[i].func, i))
            return RTERROR;
    }
    return RTNORM;
}
```

As the code sample shows, the first argument to `acedRegFunc()` is the function pointer (named after the function handler defined in the source code), and not the external function name defined by `acedDefun()` and called by AutoLISP or `acedInvoke()`. Both `acedDefun()` and `acedRegFunc()` pass the same integer function code `i`.

If a registered function is to retrieve arguments, it must do so by making its own call to `acedGetArgs()`.

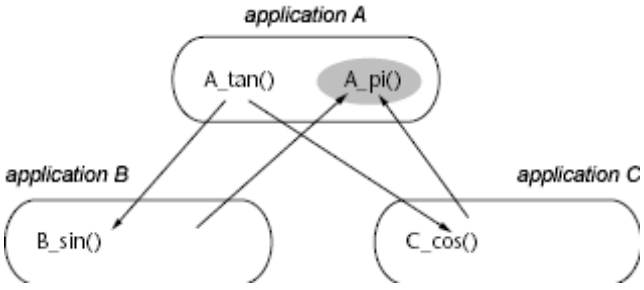
The `acedGetArgs()` call is moved to be within the function `fact()`. The result-buffer pointer `rb` is made a variable rather than an argument. (This doesn't match the call to `fact()` in the `dofun()` function elsewhere in this sample. If all external functions are registered, as this example assumes, the `dofun()` function can be deleted completely; see the note that follows this example.) The new code is shown in boldface type:

```
static int fact()
{
    int x;

        struct resbuf *rb;

        rb = acedGetArgs();

    if (rb == NULL)
        return RTERROR;
    if (rb->restype == RTSHORT) {
        x = rb->resval.rint; // Save in local variable.
    } else {
        acdbFail("Argument should be an integer.");
        return RTERROR;
    }
    if (x < 0) { // Check the argument range.
        acdbFail("Argument should be positive.");
        return RTERROR;
    } else if (x > 170) { // Avoid floating-point overflow.
        acdbFail("Argument should be 170 or less.");
        return RTERROR;
    }
    acedRetReal(rfact(x)); // Call the function itself, and
                           // return the value to AutoLISP.
    return RTNORM;
}
```

A comparable change would have to be made to `squareroot()`.

Note If an application calls `acedRegFunc()` to register a handler for every external function it defines, it can assume that these functions will be invoked by `acedInvoke()`, and it can omit the `kInvkSubrMsg` case in its `acrxEntryPoint()` function. If you design an application that requires more than a single ObjectARX code file, this technique is preferable, because it places the burden of handling function calls on the ObjectARX library rather than on the `acrxEntryPoint()` function.

If a function call starts a calling sequence that causes a function in the same application to be called with `acedInvoke()`, the latter function must be registered by `acedRegFunc()`. If the called function isn't registered, `acedInvoke()` reports an error. The following figure illustrates this situation:



In the illustration above,

- `A_tan()` invokes `B_sin()`

- `A_tan()` invokes `C_cos()`
- `B_sin()` invokes `A_pi()`
- `C_cos()` invokes `A_pi()`

where application A defines `A_tan()` and `A_pi()`, application B defines `B_sin()`, and application C defines `C_cos()`. The `A_pi()` function must be registered by `acedRegFunc()`.

To prevent `acedInvoke()` from reporting registration errors, register any external function that is meant to be called with `acedInvoke()`.

The `acedRegFunc()` function can be called also to unregister an external function. The same application must either register or unregister the function; ObjectARX prohibits an application from directly managing another application.

Topics in this section

- Handling Errors from Invoked Functions

Handling Errors from Invoked Functions

When `acedInvoke()` returns `RTNORM`, this implies that the external function was called and returned successfully. It does not imply that the external function successfully obtained a result; to obtain this information, your program must inspect the result argument. If the external function is successful and is meant to return values, result points to a result-buffer list containing one or more values. If the external function failed, the result argument is set to `NULL`. The result argument is also `NULL` if the external function doesn't return a result.

The following sample code fragment checks the return value of an external function that is expected to return one or more result values:

```
struct resbuf *xfcnlist, *xresults;
// Build the invocation list, xfcnlist.
rc = acedInvoke(xfcnlist, &xresults);
if (rc != RTNORM) {
// Couldn't call the function—report this error (or even abort).
    return BAD;
}
if (xresults == NULL) {
    // Function was called but returned a bad result.
    return BAD;
}
// Look at return results and process them.
```

Handling External Applications

ObjectARX applications can load and unload other ObjectARX applications and obtain a list

of which external applications are currently loaded, just as AutoLISP programs can (using `arxloaded`). The following call loads a program called myapp:

```
if (acedArxLoad("myapp") != RTERROR) {
    // Use acedInvoke() to call functions in "myapp".
}
```

When your program is finished with *myapp*, it can unload it by calling `acedArxUnload()`:

```
acedArxUnload("myapp");
```

The function `acedArxLoaded()` can be used to obtain the names of all currently loaded applications, as in the following code:

```
struct resbuf *rb1, *rb2;
for (rb2 = rb1 = acedArxLoaded(); rb2 != NULL; rb2 = rb2->rbnext) {
    if (rb2->restype == RTSTR)
        acutPrintf("%s\n", rb2->resval.rstring);
}
acutRelRb(rb1);
```

You can call the functions `acedArxLoaded()` and `acedArxUnload()` in conjunction with each other. The following example unloads all applications except the current one:

```
struct resbuf *rb1, *rb2;
for (rb2 = rb1 = acedArxLoaded(); rb2 != NULL;
    rb2 = rb2->rbnext) {
    if (strcmp(ads_appname, rb2->resval.rstring) != 0)
        acedArxUnload(rb2->resval.rstring);
}
acutRelRb(rb1);
```

Variables, Types, and Values Defined in ObjectARX

ObjectARX defines a few data types for compatibility with the AutoCAD environment. It also defines a number of symbolic codes for values passed by functions (or simply for general clarity). Finally, it declares and initializes a few global variables. The definitions and declarations appear in the ObjectARX header files.

Note

If an application does not adhere to the conventions imposed by the definitions and declarations described in this section, it will be difficult to read and maintain at best; at worst, it will not communicate with AutoCAD correctly. Also, future versions of ObjectARX may involve changes to the header files. Therefore, do not substitute an integer constant for its symbolic code if such a code has been defined.

Topics in this section

- General Types and Definitions
- Useful Values

- Result Buffers and Type Codes
- ObjectARX Function Result Type Codes
- User-Input Control Bit Codes

General Types and Definitions

The types and definitions described in this section provide consistency between applications and conformity with the requirements of AutoCAD. They also contribute to an application's legibility.

Topics in this section

- Real Numbers
- Points
- Transformation Matrices
- Entity and Selection Set Names

Real Numbers

Real values in AutoCAD are always double-precision floating-point values. ObjectARX preserves this standard by defining the special type `ads_real`, as follows:

```
typedef double ads_real;
```

Real values in an ObjectARX application are of the type `ads_real`.

Points

AutoCAD points are defined as the following array type:

```
typedef ads_real ads_point[3];
```

A point always includes three values. If the point is two-dimensional, the third element of the array can be ignored; it is safest to initialize it to 0.

ObjectARX defines the following point values:

```
#define X 0
#define Y 1
#define Z 2
```

Unlike simple data types (or point lists in AutoLISP), a point cannot be assigned with a single statement. To assign a pointer, you must copy the individual elements of the array, as shown in the following example:

```
newpt[X] = oldpt[X];
newpt[Y] = oldpt[Y];
newpt[Z] = oldpt[Z];
```

You can also copy a point value with the `ads_point_set()` macro. The result is the second argument to the macro.

The following sample code sets the point `to` equal to the point `from`:

```
ads_point to, from;

from[X] = from[Y] = 5.0; from[Z] = 0.0;
ads_point_set(from, to);
```

Note This macro, like the `ads_name_set()` macro, is defined differently, depending on whether or not the symbol `__STDC__` (for standard C) is defined. The standard C version of `ads_point_set()` requires that your program include `string.h`.

```
#include <string.h>
```

Because of the argument-passing conventions of the C language, points are passed by reference without the address (indirection) operator `&`. (C always passes array arguments by reference, with a pointer to the first element of the array.)

The `acedOsnap()` library function takes a point as an argument, and returns a point as a result. It is declared as follows:

```
int acedOsnap(pt, mode, result)
ads_point pt;
char *mode;
ads_point result;
```

The `acedOsnap()` function behaves like the AutoLISP `osnap` function. It takes a point (`pt`) and some object snap modes (specified in the string mode), and returns the nearest point (in `result`). The `int` value that `acedOsnap()` returns is a status code that indicates success (`RTNORM`) or failure.

The following code fragment calls `acedOsnap()`:

```
int findendpoint(ads_point oldpt, ads_point newpt)
{
    ads_point ptres;
    int foundpt;
    foundpt = acedOsnap(oldpt, "end", ptres);
    if (foundpt == RTNORM) {
        ads_point_set(ptres, newpt);
    }
    return foundpt;
}
```

Because points are arrays, `oldpt` and `ptres` are automatically passed to `acedOsnap()` by reference (that is, as pointers to the first element of each array) rather than by value. The `acedOsnap()` function returns its result (as opposed to its status) by setting the value of the `newpt` argument.

ObjectARX defines a pointer to a point when a pointer is needed instead of an array type.

```
typedef ads_real *ads_pointp;
```

Transformation Matrices

The functions `acedDragGen()`, `acedGrVecs()`, `acedNEntSelP()`, and `acedXformSS()` multiply the input vectors by the transformation matrix defined as a 4x4 array of real values.

```
typedef ads_real ads_matrix[4][4];
```

The first three columns of the matrix specify scaling and rotation. The fourth column of the matrix is a translation vector. ObjectARX defines the symbol `T` to represent the coordinate of this vector, as follows:

```
#define T 3
```

The matrix can be expressed as follows:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

The following function initializes an identity matrix.

```
void ident_init(ads_matrix id)
{
    int i, j;
    for (i=0; i<=3; i++)
        for (j=0; j<=3; j++)
            id[i][j] = 0.0;
    for (i=0; i<=3; i++)
        id[i][i] = 1.0;
}
```

The functions that pass arguments of the `ads_matrix` type treat a point as a column vector of dimension 4. The point is expressed in homogeneous coordinates, where the fourth element of the point vector is a scale factor that is normally set to 1.0. The final row of the matrix has the nominal value of [0,0,0,1]; it is ignored by the functions that pass

ads_matrix arguments. In this case, the following matrix multiplication results from the application of a transformation to a point:

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1.0 \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \bullet \begin{bmatrix} X \\ Y \\ Z \\ 1.0 \end{bmatrix}$$

This multiplication gives us the individual coordinates of the point as follows:

$$X' = M_{00}X + M_{01}Y + M_{02}Z + M_{03}(1.0)$$

$$Y' = M_{10}X + M_{11}Y + M_{12}Z + M_{13}(1.0)$$

$$Z' = M_{20}X + M_{21}Y + M_{22}Z + M_{23}(1.0)$$

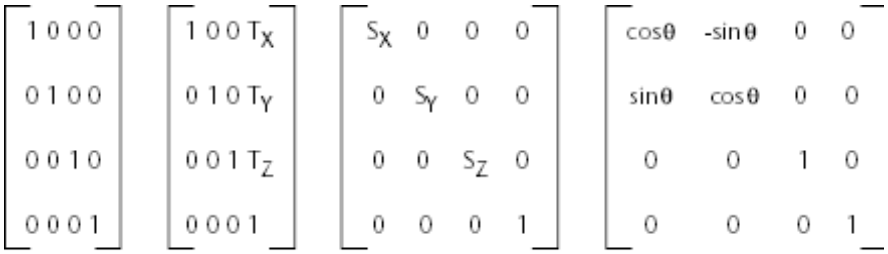
As these equations show, the scale factor and the last row of the matrix have no effect and are ignored. This is known as an affine transformation.

Note To transform a vector rather than a point, do not add in the translation vector $M_{3}M_{13}$ M_{23} (from the fourth column of the transformation matrix).

The following function implements the previous equations to transform a single point:

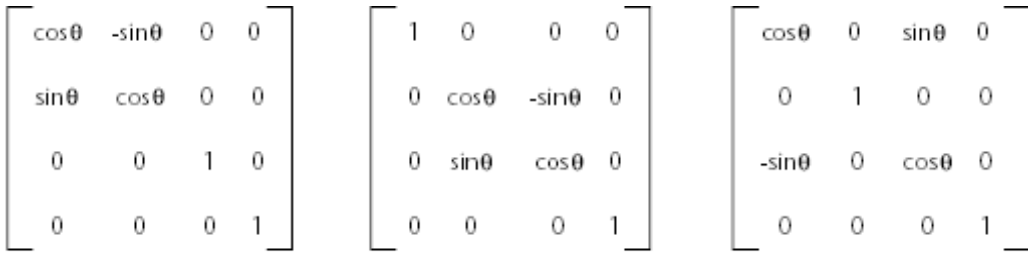
```
void xformpt(xform, pt, newpt)
ads_matrix xform;
ads_point pt, newpt;
{
    int i, j;
    newpt[X] = newpt[Y] = newpt[Z] = 0.0;
    for (i=X; i<=Z; i++) {
        for (j=X; j<=Z; j++)
            newpt[i] += xform[i][j] * pt[j];
// Add the translation vector.
        newpt[i] += xform[i][T];
    }
}
```

The following figure summarizes some basic geometrical transformations. (The values in an ads_matrix are actually ads_real, but they are shown here as integers for readability and to conform to mathematical convention.)



The acedXformSS() function—unlike the acedDragGen(), acedGrVecs(), or acedNEntSelP

() functions—requires the matrix to do uniform scaling. That is, in the transformation matrix that you pass to acedXformSS(), the elements in the scaling vector $S_X S_Y S_Z$ must all be equal; in matrix notation, $M_{00} = M_{11} = M_{22}$. Three-dimensional rotation is a slightly different case, as shown in the following figure:



rotation in the XY plane *rotation in the YZ plane* *rotation in the XZ plane*

For uniform rotations, the 3x3 submatrix delimited by [0,0] and [2,2] is orthonormal. That is, each row is a unit vector and is perpendicular to the other rows; the scalar (dot) product of two rows is zero. The columns are also unit vectors that are perpendicular to each other. The product of an orthonormal matrix and its transpose equals the identity matrix. Two complementary rotations have no net effect.

Complex transformations can be accomplished by combining (or composing) nonidentity values in a single matrix.

Note The acedTablet() function uses a 3x3 matrix to transform 2D points. The acedNEntSel() function uses a 4x3 transformation matrix that is similar to the 4x4 transformation matrix, but it treats the point as a row.

Entity and Selection Set Names

In AutoLISP, the names of entities and selection sets are pairs of long integers. ObjectARX preserves this standard by defining such names as an array type, as follows:

```
typedef long ads_name[2];
```

As with ads_point variables, ads_name variables are always passed by reference but must be assigned element by element.

You can also copy an entity or selection set name by calling the ads_name_set() macro. As with ads_point_set() and ObjectARX functions, the result is the second argument to the macro.

The following sample code sets the name newname to equal oldname.

```
ads_name oldname, newname;

if (acdbEntNext(NULL, oldname) == RTNORM)
ads_name_set(oldname, newname);
```

Note This macro, like the ads_point_set() macro, is defined differently, depending on whether or not the symbol __STDC__ (which stands for standard C) is defined. The standard C version of ads_name_set() requires your program to include string.h.

The ads_name_equal() macro compares the names in the following example:

```
if (ads_name_equal(oldname, newname))
...
```

To assign a null value to a name, call the ads_name_clear() macro, and test for a null entity or selection set name with the macro ads_name_nil().

The following sample code clears the oldname set in a previous example:

```
ads_name_clear(oldname);
```

And the following code tests whether the name is NULL:

```
if (ads_name_nil(oldname))
...
```

ObjectARX creates the following data type for situations that require a name to be a pointer rather than an array:

```
typedef long *ads_namep;
```

Useful Values

ObjectARX provides the following preprocessor definitions for use with legacy global functions:

```
#define TRUE 1
#define FALSE 0
#define EOS'\0' // String termination character
```

The PAUSE symbol, a string that contains a single backslash, is defined for the acedCommand() and acedCmd() functions, as follows:

```
#define PAUSE "\\" // Pause in command argument list
```

Note The ObjectARX library doesn't define the values GOOD and BAD, which appear as return values in the code samples throughout this guide (especially in error-handling code). You can define them if you want, or substitute a convention that you prefer.

Result Buffers and Type Codes

A general-purpose result buffer (resbuf) structure handles all of the AutoCAD data types.

Type codes are defined to specify the data types in a result buffer.

Topics in this section

- Result-Buffer Lists
- struct resbuf
- Result Type Codes Defined by ObjectARX
- DXF Group Codes

Result-Buffer Lists

Result buffers can be combined in linked lists, described later in detail, and are therefore suitable for handling objects whose lengths can vary and objects that can contain a mixture of data types. Many ObjectARX functions return or accept either single result buffers (such as `acedSetVar()`) or result-buffer lists (such as `acdbEntGet()` and `acdbTblSearch()`).

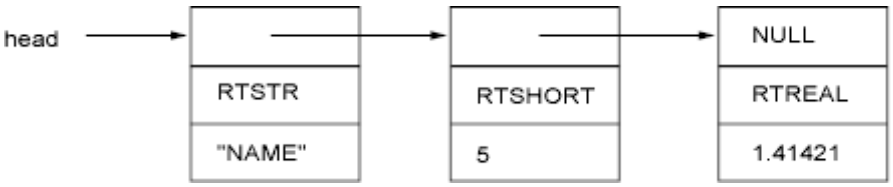
struct resbuf

The following result-buffer structure, `resbuf`, is defined in conjunction with a union, `ads_u_val`, that accommodates the various AutoCAD and ObjectARX data types, as follows:

```
union ads_u_val {
    ads_real rreal;
    ads_real rpoint[3];
    short rint; // Must be declared short, not int.
    char *rstring;
    long rlname[2];
    long rlong;
    struct ads_binary rbinary;
};
struct resbuf {
    struct resbuf *rbnext; // Linked list pointer
    short restype;
    union ads_u_val resval;
};
```

Note The `long` integer field `resval.rlong` is like the binary data field `resval.rbinary`; both hold extended entity data.

The following figure shows the schematic form of a result-buffer list:



Result Type Codes Defined by ObjectARX

The `restype` field of a result buffer is a `short` integer code that indicates which type of value is stored in the `resval` field of the buffer. For results passed to and from ObjectARX functions, ObjectARX defines the result type codes listed in the following table:

Result type codes

Code	Value	Description
RTNONE	5000	No result value
RTREAL	5001	Real (floating-point) value
RTPOINT	5002	2D point (X and Y; Z == 0.0)
RTSHORT	5003	Short (16-bit) integer
RTANG	5004	Angle
RTSTR	5005	String
RTENAME	5006	Entity name
RTPICKS	5007	Selection set name
RTORINT	5008	Orientation
RT3DPOINT	5009	3D point (X, Y, and Z)
RTLONG	5010	Long (32-bit) integer
RTVOID	5014	Void (blank) symbol
RTLB	5016	List begin (for nested list)
RTLE	5017	List end (for nested list)
RTDOTE	5018	Dot (for dotted pair)
RTNIL	5019	AutoLISP nil
RTDXF0	5020	Group code zero for DXF lists (used only with acutBuildList())
RTT	5021	AutoLISP t (true)
RTRESBUF	5023	Resbuf
RTMODELESS	5027	Interrupted by modeless dialog

DXF Group Codes

Many ObjectARX functions return the type codes defined in the preceding table. However, in results from the functions that handle entities, the `restype` field contains DXF group codes, which are described in the *AutoCAD Customization Guide*. For example, in an entity list, a `restype` field of 10 indicates a point, while a `restype` of 41 indicates a real value.

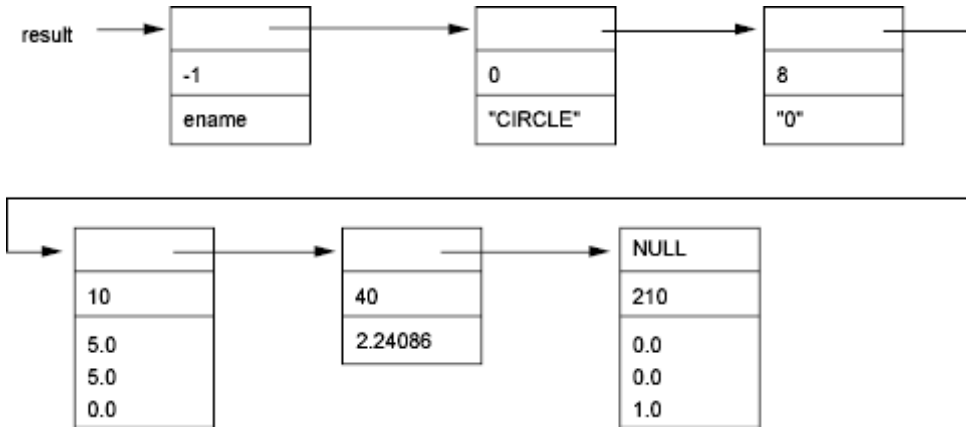
AutoCAD drawings consist of structured containers for database objects having the following components:

- A unique handle that is always enabled and that persists for the lifetime of the drawing
- An optional xdata list
- An optional persistent reactor set
- An optional ownership pointer to an extension dictionary, which owns other database objects placed in it by the application

Database objects are objects without layer, linetype, color, or any other geometric or graphical properties, and entities are derived from objects and have geometric and graphical properties.

Because DXF codes are always less than 2,000 and the result type codes are always greater, an application can easily determine when a result-buffer list contains result values (as returned by `acedGetArgs()`, for example) or contains entity definition data (as returned by `acdbEntGet()` and other entity functions).

The following figure shows the result-buffer format of a circle retrieved by `acdbEntGet()`:



The following sample code fragment shows a function, `dxftype()`, that is passed a DXF group code and the associated entity, and returns the corresponding type code. The type code indicates what data type can represent the data: `RTREAL` indicates a double-precision floating-point value, `RT3DPOINT` indicates an `ads_point`, and so on. The kind of entity (for example, a normal entity such as a circle, a block definition, or a table entry such as a viewport) is indicated by the type definitions that accompany this function:

```
#define ET_NORM 1 // Normal entity
#define ET_TBL  2 // Table
#define ET_VPORT  3 // Table numbers
#define ET_LTYPE  4
#define ET_LAYER  5
#define ET_STYLE  6
#define ET_VIEW   7
#define ET_UCS    8
#define ET_BLOCK  9
// Get basic C-language type from AutoCAD DXF group code (RTREAL,
// RTANG are doubles, RTPOINT double[2], RT3DPOINT double[3],
// RTENAME long[2]). The etype argument is one of the ET_
// definitions.
//
// Returns RTNONE if grpcode isn't one of the known group codes.
// Also, sets "inxdata" argument to TRUE if DXF group is in XDATA.
//
short dxftype(short grpcode, short etype, int *inxdata)
{
    short rbtype = RTNONE;
    *inxdata = FALSE;
    if (grpcode >= 1000) {  // Extended data (XDATA) groups
        *inxdata = TRUE;
        if (grpcode == 1071)
            rbtype = RTLONG; // Special XDATA case
        else
            grpcode %= 1000; // All other XDATA groups match.
    } // regular DXF code ranges
    if (grpcode <= 49) {
        if (grpcode >= 20) // 20 to 49
            rbtype = RTREAL;
        else if (grpcode >= 10) { // 10 to 19
            if (etype == ET_VIEW) // Special table cases
                rbtype = RTPOINT;
            else if (etype == ET_VPORT && grpcode <= 15)
                rbtype = RTPOINT;
            else // Normal point
                rbtype = RT3DPOINT; // 10: start point, 11: endpoint
        }
        else if (grpcode >= 0) // 0 to 9
            rbtype = RTSTR; // Group 1004 in XDATA is binary
        else if (grpcode >= -2)
            // -1 = start of normal entity -2 = sequence end, etc.
            rbtype = RTENAME;
        else if (grpcode == -3)
            rbtype = RTSHORT; // Extended data (XDATA) sentinel
    }
    else {
        if (grpcode <= 59) // 50 to 59
            rbtype = RTANG; // double
        else if (grpcode <= 79) // 60 to 79
            rbtype = RTSHORT;
        else if (grpcode < 210)
            ;
        else if (grpcode <= 239) // 210 to 239
            rbtype = RT3DPOINT;
        else if (grpcode == 999) // Comment
            rbtype = RTSTR;
    }
    return rbtype;
}
```

An application obtains a result-buffer list (called rb), representing an entry in the viewport symbol table, and the following C statement calls dxftype():

```
ctype = dxftype(rb->restype, ET_VPORT, &inxdata);
```

Suppose rb->restype equals 10. Then dxftype() returns RTPOINT, indicating that the entity is a two-dimensional point whose coordinates (of the type ads_real) are in rb->resval.rpoint[X] and rb->resval.rpoint[Y].

ObjectARX Function Result Type Codes

The following result type codes are the status codes returned by most ObjectARX global functions to indicate success, failure, or special conditions (such as user cancellation)on result type codes:

Library function result codes

Code	Value	Description
RTNORM	5100	User entered a valid value
RTERROR	-5001	The function call failed
RTCAN	-5002	User entered ESC
RTREJ	-5003	AutoCAD rejected the request as invalid
RTFAIL	-5004	AutoLISP communication failed
RTKWORD	-5005	User entered a keyboard or arbitrary text
RTINPUTTRUNCATED	-5008	Input didn't fit in the buffer

The meanings of these codes, summarized in the table, are as follows:

RTNORM

The library function succeeded.

RTERROR

The library function did not succeed; it encountered a recoverable error.

The RTERROR condition is exclusive of the following special cases:

RTCAN

The AutoCAD user entered ESC to cancel the request. This code is returned by the user-input (acedGetxxx) functions and by the following functions: acedCommand, acedCmd, acedEntSel, acedNEntSelP, acedNEntSel, and acedSSGet.

RTREJ

AutoCAD rejected the operation as invalid. The operation request may be incorrectly formed, such as an invalid `acdbEntMod()` call, or it simply may not be valid for the current drawing.

RTFAIL

The link with AutoLISP failed. This is a `fatal` error that probably means AutoLISP is no longer running correctly. If it detects this error, the application should quit. (Not all applications check for this code, because the conditions that can lead to it are likely to hang AutoCAD, anyway.)

RTKWORD

The AutoCAD user entered a keyword or arbitrary input instead of another value (such as a point). The user-input `acedGetxxx()` functions, as well as `acedEntSel`, `acedEntSelP`, `acedNEntSel`, and `acedDragGen`, return this result code.

Note Not all ObjectARX global functions return these status codes; some return values directly. Also, the user-input (`acedGetxxx`, `acedEntSel`, `acedEntSelP`, `acedNEntSel`, and `acedDragGen`) functions can return the `RTNONE` result type code, and `acedDragGen()` indicates arbitrary input by returning `RTSTR` instead of `RTKWORD`.

User-Input Control Bit Codes

The user-input control bit codes listed in the following table are passed as the first argument to the `acedInitGet()` function to control the behavior of user-input functions `acedGetxxx`, `acedEntSel`, `acedNEntSelP`, `acedNEntSel`, and `acedDragGen`:

User-input control bit codes

Code	Description
RSG_NONULL	Disallow null input
RSG_NOZERO	Disallow zero values
RSG_NONEG	Disallow negative values
RSG_NOLIM	Do not check drawing limits, even if LIMCHECK is on
RSG_DASH	Use dashed lines when drawing rubber-band line or box
RSG_2D	Ignore *Z* coordinate of 3D points (acedGetDist() only)
RSG_OTHER	Allow arbitrary input (whatever the user types)

Lists and Other Dynamically Allocated Data

The resbuf structure includes a pointer field, `rbnext`, for linking result buffers into a list. Result buffers can be allocated statically by declaring them in the application. You do this when only a single result buffer is used (for example, by `acedGetVar()` and `acedSetVar()`) or when only a short list is needed. But longer lists are easier to handle by allocating them dynamically, and lists returned by ObjectARX functions are always allocated dynamically. One of the most frequently used functions that returns a linked list is `acedGetArgs()`.

Evaluating External Functions shows the AutoLISP calling format of an external subroutine that takes arguments of three distinct types: a string, an integer, and a real value:

```
(doit pstr iarg rarg)
```

The following code segment shows how to implement a function with such a calling sequence. The sample function checks that the argument list is correct and saves the values locally before operating on them (operations are not shown). The example assumes that a previous call to `acedDefun()` has assigned the external subroutine a function code of 0, and that all functions defined by this application take at least one argument:

```
// Execute a defined function.
int dofun()
{
    struct resbuf *rb;
    char str[64];
    int ival, val;
    ads_real rval;
    ads_point pt;
// Get the function code.
    if ((val = acedGetFuncode()) == RTERROR)
        return BAD; // Indicate failure.
    // Get the arguments passed in with the function.
    if ((rb = acedGetArgs()) == NULL)
        return BAD;
    switch (val) { // Which function is called?
    case 0: // (doit)
        if (rb->restype != RTSTR) {
            acutPrintf("\nDOIT called with %d type.",
                rb->restype);
            acutPrintf("\nExpected a string.");
            return BAD;
        }
// Save the value in local string.
        strcpy(str, rb->resval.rstring);
// Advance to the next result buffer.
        rb = rb->rbnext;
        if (rb == NULL) {
            acutPrintf("\nDOIT: Insufficient number of
                arguments.");
            return BAD;
        }
        if (rb->restype != RTSHORT) {
            acutPrintf("\nDOIT called with %d type.",
                rb->restype);
            acutPrintf("\nExpected a short integer.");
            return BAD;
        }
// Save the value in local variable.
        ival = rb->resval.rint;
// Advance to the last argument.
        rb = rb->rbnext;
        if (rb == NULL) {
            acutPrintf("\nDOIT: Insufficient number of
                arguments.");
            return BAD;
        }
        if (rb->restype != RTREAL) {
            acutPrintf("\nDOIT called with %d type.",
                rb->restype);
            acutPrintf("\nExpected a real.");
            return BAD;
        }
// Save the value in local variable.
        rval = rb->resval.rreal;
// Check that it was the last argument.
        if (rb->rbnext != NULL) {
                acutPrintf("\nDOIT: Too many arguments.");
            return BAD;
        }
// Operate on the three arguments.
        . . .
        return GOOD;  // Indicate success
        break;
```

Note This example is exceptional in one respect: `acedGetArgs()` is the only ObjectARX global function that returns a linked list that the application does not have to explicitly release. The following section describes the usual way of managing the memory needed for lists.

Topics in this section

- Result-Buffer Memory Management

Result-Buffer Memory Management

The main difference between result-buffer lists and comparable AutoLISP result lists is that an ObjectARX application must explicitly manage the lists that it creates and uses. Whether an application creates a list or has one passed to it, it is the application's responsibility to release the result buffers that it allocates. ObjectARX has no automatic garbage collection as AutoLISP does. The application must call the library function `acutRelRb()` to release dynamically allocated result buffers when the application is finished with them.

The `acutRelRb()` function releases the entire list that follows the specified result buffer, including the specified (head) buffer itself and any string values that the buffers in the list point to. To release a string without removing the buffer itself, or to release a string belonging to a static result buffer, the application must call the standard C library function `free()`.

Warning Do not write data to a dynamic location that hasn't been allocated with direct calls to `malloc()` or with the ObjectARX library (including `acutNewRb()`). This can corrupt data in memory. Conversely, calling `free()` or `acutRelRb()` to release data that was allocated statically—in a static or automatic variable declaration—also can corrupt memory. Inserting a statically allocated variable, such as a string, into a result-buffer list causes your program to fail when you release the list with `acutRelRb()`.

Sample calls to `acutRelRb()` appear in several of the code examples in the following sections.

Topics in this section

- List Creation and Deletion
- AutoLISP Lists
- Entity Lists with DXF Codes in ObjectARX
- Command and Function Invocation Lists

List Creation and Deletion

An ObjectARX application can dynamically allocate a single result buffer by calling `acutNewRb()`. The call to `acutNewRb()` must specify the type of buffer to allocate; `acutNewRb()` automatically initializes the buffer's `restype` field to contain the specified type code.

The following sample code fragment allocates a result buffer to contain a three-dimensional point and then initializes the point value:

```
struct resbuf *head;
if ((head=acutNewRb(RT3DPOINT)) == NULL) {
    acdbFail("Unable to allocate buffer\n");
    return BAD;
}
head->resval.rpoint[X] = 15.0;
head->resval.rpoint[Y] = 16.0;
head->resval.rpoint[Z] = 11.18;
```

If the new result buffer is to contain a string, the application must explicitly allocate memory to contain the string:

```
struct resbuf *head;
if ((head=acutNewRb(RTSTR)) == NULL) {
    acdbFail("Unable to allocate buffer\n");
    return BAD;
}
if ((head->resval.rstring = malloc(14)) == NULL) {
    acdbFail("Unable to allocate string\n");
    return BAD;
}
strcpy(head->resval.rstring, "Hello, there.");
```

Memory allocated for strings that are linked to a dynamic list is released when the list is released, so the following call releases all memory allocated in the previous example:

```
acutRelRb(head);
```

To release the string without releasing the buffer, call `free()` and set the string pointer to `NULL` as shown in the following example:

```
free(head->resval.rstring);
head->resval.rstring = NULL;
```

Setting `resval.rstring` to `NULL` prevents a subsequent call to `acutRelRb()` from trying to release the string a second time.

If the elements of a list are known beforehand, a quicker way to construct it is to call `acutBuildList()`, which takes a variable number of argument pairs (with exceptions such as `RTLB`, `RTLE`, `-3`, and others) and returns a pointer to a list of result buffers that contains the specified types and values, linked together in the order in which they were passed to `acutBuildList()`. This function allocates memory as required and initializes all values. The last argument to `acutBuildList()` must be a single argument whose value is either zero or `RTNONE`.

The following sample code fragment constructs a list that consists of three result buffers. These contain a real value, a string, and a point, in that order:

```
struct resbuf *result;
ads_point pt1 = {1.0, 2.0, 5.1};
result = acutBuildList(
    RTREAL, 3.5,
    RTSTR, "Hello, there.",
    RT3DPOINT, pt1,
    0 );
```

If it cannot construct the list, `acutBuildList()` returns `NULL`; otherwise, it allocates space to contain the list. This list must be released by a subsequent call to `acutRelRb()`:

```
if (result != NULL)
    acutRelRb(result);
```

AutoLISP Lists

The `acutBuildList()` function is called in conjunction with `acedRetList()`, which returns a list structure to AutoLISP.

The following sample code fragment passes a list of four points:

```
struct resbuf *res_list;
ads_point ptarray[4];
// Initialize the point values here.
.
.
.
res_list = acutBuildList(
    RT3DPOINT, ptarray[0],
    RT3DPOINT, ptarray[1],
    RT3DPOINT, ptarray[2],
    RT3DPOINT, ptarray[3], 0);
if (res_list == NULL) {
    acdbFail("Couldn't create list\n");
    return BAD;
}
acedRetList(res_list);
acutRelRb(res_list);
```

Dotted pairs and nested lists can be returned to AutoLISP by calling `acutBuildList()` to build a list created with the special list-construction type codes. These codes are needed only for complex lists. For ordinary (that is, one-dimensional) lists, `acedRetList()` can be passed a simple list of result buffers, as shown in the previous example.

Note A list returned to AutoLISP by `acedRetList()` can include only the following result type codes: `RTREAL`, `RTPOINT`, `RTSHORT`, `RTANG`, `RTSTR`, `RTENAME`, `RTPICKS`, `RTORINT`, `RT3DPOINT`, `RTLB`, `RTLE`, `RTDOTE`, `RTNIL`, and `RTT`. (Although there is an `RTNIL` return code, if you are returning only a `nil` list, you can call `acedRetNil()`). It can contain result types of `RTLONG` if the list is being returned to another ObjectARX application.

Use of the list-construction type codes is simple. In the `acutBuildList()` call, a nested list

is preceded by the result type code RTLB (for List Begin) and is followed by the result type code RTLE (for List End). A dotted pair can also be constructed. Dotted pairs also begin with RTLB and end with RTLE; the dot is indicated by the result type code RTDOTE, and appears between the two members of the pair.

Note This is a change from earlier versions. Applications that receive a dotted pair from AutoLISP no longer have to modify the format of the dotted pair before returning it with acedRetList(). (The earlier order, with RTDOTE at the end, is still supported.)

Warning The acutBuildList() function does not check for a well-formed AutoLISP list. For example, if the RTLB and RTLE codes are not balanced, this error is not detected. If the list is not well formed, AutoLISP can fail. Omitting the RTLE code is guaranteed to be a fatal error.

The following sample code fragment constructs a nested list to return to AutoLISP:

```
res_list = acutBuildList(
    RTLB, // Begin sublist.
    RTSHORT, 1,
    RTSHORT, 2,
    RTSHORT, 3,
    RTLE, // End sublist.
    RTSHORT, 4,
    RTSHORT, 5,
    0);
if (res_list == NULL) {
    acdbFail("Couldn't create list\n");
    return BAD;
}
acedRetList(res_list);
acutRelRb(res_list);
```

The list that this example returns to AutoLISP has the following form:

((1 2 3) 4 5)

The following code fragment constructs a dotted pair to return to AutoLISP:

```
res_list = acutBuildList(
    RTLB, // Begin dotted pair.
    RTSTR, "Sample",
    RTDOTE,
    RTSTR, "Strings",
    RTLE, // End dotted pair.
    0);
if (res_list == NULL) {
    acdbFail("Couldn't create list\n");
    return BAD;
}
acedRetList(res_list);
acutRelRb(res_list);
```

The list that this example returns to AutoLISP has the following form:

(("Sample" . "Strings"))

Note In AutoLISP, dotted pairs associate DXF group codes and values. In an ObjectARX application this is unnecessary, because a single result buffer contains both the group code (in its restype field) and the value (in its resval field). While ObjectARX provides the list-

construction type codes as a convenience, most ObjectARX applications do not require them.

Entity Lists with DXF Codes in ObjectARX

As previously mentioned, lists with DXF group codes represent AutoCAD entities. The `acutBuildList()` function constructs such lists. To construct an entity, call both `acutBuildList()` and `acdbEntMake()`.

Note Entity definitions begin with a zero (`0`) group that describes the entity type. Because lists passed to `acutBuildList()` are terminated with `0` (or `RTNONE`), this creates a conflict. The special result type code `RTDXF0` resolves the conflict. Construct the zero group in DXF lists passed to `acutBuildList()` with `RTDXF0`. If you attempt to substitute a literal zero for `RTDXF0`, `acutBuildList()` truncates the list.

The following sample code fragment creates a DXF list that describes a circle and then passes the new entity to `acdbEntMake()`. The circle is centered at (4,4), has a radius of 1, and is colored red:

```
struct resbuf *newent;
ads_point center = {4.0, 4.0, 0.0};
newent = acutBuildList(
    RTDXF0, "CIRCLE",
    62, 1, // 1 == red
    10, center,
    40, 1.0, // Radius
    0 );
if (acdbEntMake(newent) != RTNORM) {
    acdbFail("Error making circle entity\n");
    return BAD;
}
```

Command and Function Invocation Lists

Finally, `acutBuildList()` is called in conjunction with `acedCmd()`, which takes a result-buffer list to invoke AutoCAD commands, and with `acedInvoke()`, which invokes an external function from a different ObjectARX application.

The following sample code fragment calls `acutBuildList()` and `acedInvoke()` to invoke the RESET command defined by the sample application gravity.c:

```
struct resbuf *callist, *results = NULL;
callist = acutBuildList(RTSTR, "c:reset", 0);
if (acedInvoke(callist, &results) == RTERROR)
    acdbFail("Cannot run RESET -- GRAVITY program may not
    be loaded\n");
acutRelRb(callist);
acutRelRb(results);
```

Extended Data Exclusive Data Types

Extended data (xdata) can include binary data, organized into variable-length chunks. These are handled by the ads_binary structure, as follows:

```
struct ads_binary { // Binary data chunk structure
    short clen; // Length of chunk in bytes
    char *buf; // Binary data
};
```

The value of the clen field must be in the range of 0 to 127. If an application requires more than 127 bytes of binary data, it must organize the data into multiple chunks.

With Release 13, the DXF representation of a symbol table can include extended entity data. Xdata is returned as a handle.

Note There is no mechanism for returning binary data to AutoLISP. Binary chunks can be passed to other external functions by means of acedInvoke(), but only when they belong to groups (1004) within an entity's extended data. You cannot pass isolated binary chunks.

Xdata can also include long integers. The ads_u_val union of the resval field of a result buffer includes both an ads_binary and a long member for handling extended entity data.

Note There is no mechanism for returning a long integer to AutoLISP. Long integers can be passed to other external functions by means of acedInvoke(), but only when they belong to groups (1071) within an entity's extended data. In AutoLISP, 1071 groups are maintained as real values.

Text String Globalization Issues

AutoCAD Release 13 was enhanced with localization support to make AutoCAD more suitable for international customers. With this support, an AutoCAD user can enter commands in local non-English languages, and the display shows messages in the local language. The support for multiple-language character sets involves out-of-code-page characters.

Sometimes system code page strings in a .dwg file have out-of-code-page characters to display messages in another language. These characters have no normal representation in the character set of the native system. The "\U+XXXX" and "\M+XXXX" escape sequences represent these special characters in the system code page strings. The XXXX is a sequence of four hexadecimal digits that specify either the Unicode (single-character encoding) identifier or Multibyte Interchange Format (MIF) of the encoded character.

As part of Autodesk's globalization effort, the following preexisting ObjectARX functions have been changed to improve the handling of drawings created with various language versions of AutoCAD:

acdbXdSize

Returns the number of bytes of memory needed for a list of extended entity data.

acdbXdRoom

Returns the number of bytes of memory that an entity has available for extended data.

These functions count out-of-code-page characters differently.

The `acdbXdSize()` and `acdbXdRoom()` functions now recognize "`\U+XXXX`" as 1 byte, but other ObjectARX functions recognize "`\U+XXXX`" as 7 bytes. The Asian version of AutoCAD recognizes "`\M+XXXX`" as 2 bytes.

Note

ObjectARX applications that make explicit assumptions about the limit of the string length of symbol table names and TEXT entities are affected by out-of-code-page characters.

AutoCAD Command Prompt Standard

This section provides guidelines for implementing the AutoCAD command line interface for your application.

Topics in this section

- Command Line Interface

Command Line Interface

The command line interface provides an important method for controlling AutoCAD. For many users, it is the primary input method. The command line is also used for displaying and selecting command options.

The command line interface prompts the user with instructions, which may include a list of options, a list of current settings, and a default value. Instructions can be as simple as

Select objects:

or as complex as

Attach/Rotation/Style/Height/Direction/Width/2Points/<Other corner>:

The following guidelines should be used when implementing your application's AutoCAD command line interface.

Note

Changes to command line prompts should not affect the functionality of scripts or LISP compatibility of the existing command set.

Topics in this section

- General Format
- Format for the Current Value Line
- Format for the Command Line Prompt
- Examples of Command Line Prompts
- Limitations and Compatibility
- Context Menu Population

General Format

The command line interface should use a standard format. The general format or syntax of the command line should be as follows:

Description: Setting1=Value Setting2=Value Setting3=Value

Current instruction (descriptive info) or [Option1/oPtion2/opTion3/...] <default option>:

The optional command line that displays current settings is called the current value line. The required command line that provides specific instructions and displays available options is called the command line prompt.

The command line parser uses a set of special characters to correctly populate the context menus. These special characters cannot be used anywhere in the command line interface except as noted below.

Special characters in the command line interface

Symbol	Name	Function in the command prompt
:	Colon	Used to end the prompt string or after the description part of the current value line
[	Left square bracket	Indicates the start of command options
]	Right square bracket	Indicates the end of command options
<	Left angle bracket	Indicates the start of the default option
>	Right angle bracket	Indicates the end of the default option

Option keywords must be enclosed in square brackets and separated by forward slashes. If a set of parentheses is included in an option string, the parentheses must contain that entire option string as it appears in English.

There can be more than one word between slashes, but only one word must contain one or more capitalized letters that indicate the keyboard shortcut. Multiple capitalized letters must be grouped together and can appear at any location within a word. If a number appears before a capitalized letter, it is also part of the keyboard shortcut. The word that contains the capitalized letters is, by definition, the keyword. Users can enter the entire keyword or the keyboard shortcut to issue an option.

Most commands require one command line prompt. Some will be preceded by a current value line. In some cases, a prompt is split between two lines, or two prompts are issued for the same command. For example, the prompt for ZOOM looks like this:

Command: zoom

Specify corner of window, enter a scale factor (nX or nXP), or

[All/Center/Dynamic/Extents/Previous/Scale/Window/Object] <real time>:

In this case, one prompt is split, which results in two lines.

The EXTEND command has a value line and two prompts.

Command: extend

Current settings: Projection=UCS, Edge=None

Select boundary edges ...

Select objects or <select all>:

The default setting for the command line window is three lines; therefore, prompts for native commands are no more than three lines. External developers are advised to adhere to this limit as well.

Prompts should be designed to wrap before the 80th character. The line break should occur at one of the following locations:

- At a space in the current instruction portion of the prompt
- Following a slash in the option list
- Between the option list and the default

No other wrapping locations are acceptable.

Our target display for measuring the above conditions is an 800 x 600 monitor.

Format for the Current Value Line

Some commands benefit by displaying the current value of certain system variables or other settings.

The following examples show unrevised versions of command prompts:

Command: fillet

(TRIM mode) Current fillet radius = 0.000

Polyline/Radius/Trim/<Select first object>:

Select second object:

Command: attdef

Attribute modes Invisible:N Constant:N Verify:N Preset:N

Enter (ICVP) to change, RETURN when done:

A command line that displays system variables or other settings is called the current value line. The format of the current value line is as follows:

Description: Setting1=Value Setting2=Value Setting3=Value

The current value may begin with a description. The description should be separated from the settings by a colon followed by two spaces. Each setting should be described clearly without abbreviations or abridgments. The equal sign may not be preceded or followed by spaces. Each setting should be separated from the next setting by two spaces.

If the setting can be changed using an option in the command line prompt, use the same word in describing the setting.

The following example shows the revised version for the -ATTDEF command.

Command: -attdef

Invisible=N Constant=N Verify=N Preset=N Lock position=Y Annotative=N Multiple line=N

Enter an option to change [Invisible/Constant/Verify/Preset/Lock position/Annotative/Multiple lines] <done>:

If the setting requires a system variable to change, use the name of the system variable. For example:

Command: edgesurf

Current wire frame density: SURFTAB1=6 SURFTAB2=6

Select object 1 for surface edge:

The normal command line prompt should immediately follow the current value line without an extra carriage return so as to be visible in a three-line command line window.

Format for the Command Line Prompt

The required command line prompt provides the user with the current instruction, descriptive information, a list of options, and the default option. It uses the following format:

Current instruction (descriptive info) [or Option1/oPtion2/opTion3/...] <default option>:

Topics in this section

- Current Instruction
- Descriptive Information
- List of Options
- Default Option

Current Instruction

The current instruction tells the user what type of input is required and what values are acceptable. The current instruction should be as specific as possible. For example, "Enter a height..." is preferable to "Enter a number...." The current instruction should follow a specific format of wording and grammar.

The current instruction should use a verb as the first word. (This is in English; other languages should consistently place the verb where it is syntactically correct for that language.) The verb provides a standardized meaning as to the type of input that is required.

Standardized verb usage

Verb	Meaning
Select	Pick objects on the screen using your pointing device.
Enter	Enter a value at the command line.
Specify	Pick a point on the screen or enter a value at the command line.

It might appear that the word "Pick" is more appropriate than "Specify," especially when the only allowable input is a point. However, a point can be entered by picking it on screen, entering it at the command line, or by using an object snap. "Specify" is, therefore, preferable because of its broader meaning.

In certain cases, other verbs may be more appropriate for the situation. For example the AUDIT command uses:

Fix errors that are detected? [Yes/No] <Y>:

Wherever possible, the word immediately after the verb should help the user determine what kind of value is acceptable. "Enter height of target box..." is preferable to "Enter target box height...."

Examples include:

Specify a point..., Specify a start point..., Specify a base point..., Specify next point..., Specify an end point...

Specify a distance..., Specify a height..., Specify a scale...,

Specify an angle..., Specify a start angle..., Specify an end angle...

Enter a value... for a range of numbers

Enter a positive number..., Enter a negative number..., Enter a non-zero number...

Enter a file name

Enter a ... name when only one name (string) is valid

Enter a name ... list when you can enter a single name or a comma delimited list of names with wild cards

Descriptive Information

Any required descriptive information should be enclosed in parentheses. Descriptive information is used to provide further hints for acceptable input, such as valid ranges, angular direction, or suffixes. It should be avoided where possible.

Example of descriptive information:

Command: aperture

Enter height of object snap target (1-50 pixels) <current value>:

List of Options

Following the current instruction, the command line prompt may display command options. The current instruction describes drawing area operations or immediate input of values. The command options describe toggles, or command branches. These options should be enclosed in square brackets and separated by slashes. If appropriate, the word "or" may precede the bracket. The bracket, slashes, and the word "or" should be suitably localized.

The format for options is as follows:

[Option1/oPtion2/opTion3...]

The brackets are designed to help the user to identify the available command branches. They are also used by AutoCAD to process the keywords for presenting in the context (right-click) menu while the options are displayed on the command line.

Each option keyword must have a shortcut key, indicated in uppercase and unique to that string of options. When necessary, an option may have two keys used together to indicate that option.

Example:

Command: -rename

Enter object type to rename

[Block/Dimstyle/LAyer/LType/Material/multileadeRstyle/Style/Tablestyle/Ucs/VIew/Vport]:

Capitalization of options should follow standard user interface guidelines as used in menu accelerators (single mnemonic characters, consonants preferred to vowels, and so on) for new commands. Existing commands should retain their current shortcut keys.

Options should be listed or grouped by function. A suggested logical order would be most-often-used option, options related to that one, other options that can be grouped, and exit (if necessary).

Example:

Command: grid

Specify grid spacing(X) or [ON/OFF/Snap/Major/aDaptive/Limits/Follow/Aspect] <0.5000>:

Only when no logical order is apparent are the options listed in alphabetical order. This would be true in cases where all the options are equally likely to be used, and do not bear any relationships to each other.

Example:

Command: zoom

Specify corner of window, enter a scale factor (nX or nXP), or [All/Center/Dynamic/Extents/Previous/Scale/Window/Object] <real time>:

Because option keywords for native commands are presented in the same order regardless of language, alphabetic organization tends to break down when the command is translated.

Default Option

For many commands a default option is available when the Enter key is pressed. If there is no default option or value, pressing the Enter key exits from the command. The format for the default is as follows:

<default>

The text within the angle brackets should provide some hint as to the use of that option, such as <width>, not <RETURN>. The default option should not be "eXit" except in command line prompts that are nested.

If a default value is passed to the current option when the ENTER key is pressed, then that value should be enclosed in angle brackets: <??>. There should be only one set of angle brackets per prompt.

Example:

Command: cone

Specify center point of cone or [Elliptical] <0,0,0>:

Examples of Command Line Prompts

The following examples show previous and revised versions of AutoCAD command line prompts.

3DFACE

Previous:

Command: 3dface

First point:

Second point:

Third point:

Fourth point:

Third point:

Revised:

Command: 3dface

Specify first point or [Invisible]:

Specify second point or [Invisible]:

Specify third point or [Invisible] <exit>:

Specify fourth point or [Invisible] <create three-sided face>:

Specify third point or [Invisible] <exit>:

DIMANGULAR

Previous:

Command: dimangular

Select arc, circle, line, or RETURN:

Angle vertex:

First angle endpoint:

Second angle endpoint:

Dimension arc line location (Text/Angle):

Revised:

Command: dimangular

Select arc, circle, line, or <specify vertex>:

Specify angle vertex:

Specify first angle endpoint:

Specify second angle endpoint:

Specify dimension arc line location or [Mtext/Text/Angle/Quadrant]:

Limitations and Compatibility

These guidelines may cause some command prompts to be longer than they were previously; some prompts may occupy two lines.

Exceptions may have to be made for command prompts if adherence to the guidelines results in an awkwardly phrased prompt.

Script compatibility with previous versions of AutoCAD should be maintained. If option lists are shortened, the previously valid options should continue to be valid, even if they are not displayed. Option keywords and accelerators for existing options should not be changed. New options may be added, but all existing options should continue to exist and use the same shortcuts. Command syntax remains the same.

Context Menu Population

The context menu will be populated by parsing the words within square brackets of the command line options.

Starting in AutoCAD 2000, the context (or right-click) menus intercept the strings going to the command line, read the options that are between the square brackets, and present each option between slash (/) characters as a selectable item on the menu. In addition, the default option (between angle brackets) will be presented to the user as a context menu item.

Because of this dependency, developers should follow the formats as outlined in this specification if they want to take advantage of this functionality. Prompts should also be structured so that they don't present a problem when listed in a pop-up menu.