ARX Programming Environment

(from ARX Developer's Guide, Autodesk, Inc. 1996)

A Brief Outline

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

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

ARX Libraries

The ARX environment consists of the following libraries:

AcRx -- Classes used for binding an application and for run-time class registration and identification.

AcEd -- Classes for registering native commands and for system event notification.

AcDb -- AutoCAD database classes.

AcGi -- Graphics interface for rendering AutoCAD entities.

AcGe -- Utility library for common linear algebra and geometric objects.

ADS -- A C library used to create AutoCAD applications. ARX applications typically use this library for operations such as entity selection, selection set manipulation, and data acquisition. See the ADS Developer's Guide.

AutoLISP, ADS, and ARX

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

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

- **The ARX programming environment** differs from the ADS and AutoLISP programming environments in a number of ways. The most important difference is that an ARX application is a dynamic link library (DLL) that shares AutoCAD's address space and makes direct function calls to

AutoCAD, avoiding the costly overhead of IPC. Applications that communicate frequently with AutoCAD run faster in the ARX environment than in the ADS or AutoLISP environments.

In addition to speed enhancements, you can add new classes to the ARX program environment and export them for use by other programs. ARX entities you create are virtually indistinguishable from built-in AutoCAD entities. You can also extend ARX protocol by adding functions at run time to existing AutoCAD classes.

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

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

- **Other Differences**

Registering Commands

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

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

The commands can be evaluated through AutoLISP or the ads_invoke() facility

The commands cannot be invoked using the AutoLISP command function or the ads_command() function

The opposite is true for commands registered through acedRegCmds():

The commands are not known to AutoLISP or the ads_invoke() facility.

The commands can be invoked using the AutoLISP command function or the ads_command() function.

Entry Points

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

Comparing ADS and ARX Function Calls

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

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

```
}
```

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

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

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

```
void
changeLayerADS(ads_name entityName,
    const char* pNewLayerName)
{
    struct resbuf *pRb, *pTempRb;
    pRb = ads_entget(entityName);

    // No need to check for rb == NULL since all
    // entities have a layer.
    //
    for (pTempRb = pRb; pTempRb->restype != 8;
        pTempRb = pTempRb->rbnext)
    { ; }

    free(pTempRb->resval.rstring);
    pTempRb->resval.rstring
        = (char*) malloc(strlen(pNewLayerName) + 1);
    strcpy(pTempRb->resval.rstring, pNewLayerName);
    ads_entmod(pRb);
    ads_relrb(pRb);
    ads_retvoid();

}
```

Here is the AutoLISP code to do the same thing:

```
(defun asdk_changeLayerLISP(ename newLayer / eList)

    (setq eList (entget ename))

    ; substitute the new layer name for the old
    ;
    (setq eList
        (subst (cons 8 newLayer) (assoc 8 eList) eList))

    ; Modify the entity's data in drawing to reflect
    ; the changed layer
    ;
    (entmod eList)
    (princ)

)
```

The following chart compares the AutoLISP, ADS, ADS-Rx, and ARX programming interfaces with respect to speed, exposure, power, and programming expertise required to use each API. The "exposure" parameter indicates the possible severity of your programming errors. Although the ARX interface is the

most powerful of the four APIs compared here, it also offers the greatest potential for serious programming errors, such as corrupting AutoCAD data structures. The other programming environments require proportionately less programming expertise, but also provide less power and scope.

Run-Time Type Identification

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

desc(), a static member function, returns the class descriptor object of a particular (known) class

cast(), a static member function, returns an object of the specified type, or NULL if the object is not of the required class (or a derived class)
is **KindOf()** returns whether an object belongs to the specified class (or a derived class)
is **A()** returns the class descriptor object of an object whose class is unknown

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

AcRxClass* is A() const;

When you already know what class an object is, you can use the desc() function to obtain the class descriptor object:

static AcRxClass* desc();

The following example looks for instances of AcDbEllipse, or any class derived from it, using is KindOf() and the AcDbEllipse::desc() static member function.

```
AcDbEntity* curEntity = somehowGetAndOpenAnEntity();

if (curEntity->isKindOf(AcDbEllipse::desc())) {
   // Got some kind of AcDbEllipse instance.

}
```

This example shows another way of looking for instances of AcDbEllipse, or any class derived from it, using the AcDbEllipse::cast() static member function.

```
AcDbEllipse* ellipseEntity = AcDbEllipse::cast(curEntity);

if (ellipseEntity != NULL) {
   // Got some kind of AcDbEllipse instance.

}
```

The following example looks for instances of AcDbEllipse, but not instances of classes derived from AcDbEllipse, using isA() and AcDbEllipse::desc().

```
if (curEntity->isA() == AcDbEllipse::desc()) {
   // Got an AcDbEllipse, no more, no less.
```