AUTODESK **UNIVERSITY** 2015

SD9660

Creating AutoCAD Cross-Platform Plug-ins

Fernando Malard
ofcdesk, LLC

Learning Objectives

- Learn how to create C++ cross-platform core
- Learn how to create code with ObjectARX technology for Windows
- Learn how to create code with ObjectARX technology for Mac
- Learn how to use Microsoft Visual Studio and Mac OSX XCode

Description

This class will present strategies used to create cross-platform plug-ins, taking advantage of C++ language in order to create core source code that could be used by AutoCAD software for Windows and by AutoCAD for Mac software. We will then demonstrate how to consume this shared core code into each platform, taking advantage of each specific user interface feature (Microsoft MFC, Microsoft .NET inside Windows, and Cocoa inside Mac OSX).

Your AU Experts

Fernando Malard is a civil engineer who has worked with AutoCAD software and ObjectARX technology since 1996 and with Revit software since 2009. He has also been an Autodesk Developer Network member since 1997. He has worked on several AutoCAD and Revit applications for civil engineering, architecture, interior design, and geographic information system (GIS) using ObjectARX technology, C++, Microsoft .NET, JavaScript, Cocoa, and databases. Fernando has had extensive experience teaching AutoCAD, Revit, C++, Microsoft Foundation Class, Microsoft .NET, and ObjectARX technology over the last years. Today he continues to apply his skills to the design and implementation of complex Industry Solutions. Fernando has also worked with Autodesk User Group International (AUGI) communities, and he maintains a blog about ObjectARX technology. He holds a master's degree in structural engineering from the Federal University of Minas Gerais, Brazil.

Introduction

In the past, Autodesk used to support way more platforms than Windows. Old versions of AutoCAD (R12 and R13) used to run into a Macintosh computer but Autodesk stopped its support back in 1994. Since 2010 Mac OSX support was back starting with AutoCAD 2011 for Mac and lately with AutoCAD 2016 for Mac OSX. The AutoCAD for Mac UI has nothing to do with the Windows version due an infinite number of reasons but there is a huge portion of the code written in C++ that is shared between the two platforms. The shared source code includes some hard work done to support 64-bit architecture (the only supported by AutoCAD for Mac) and also to make it way smoother to port applications from Windows. Things widely used into AutoCAD for Windows plugins like **MFC** classes like **CString** where wisely supported via tricky support libraries created by Autodesk. These support libraries avoided the excessive use of **#ifdef** compile code switches to accommodate dynamic compilation changes into a source code meant to work into both Platforms.

More than a simple port to Mac OSX, AutoCAD for Mac feels like a native application with some remarkable features not present into the Windows version. Even though, the overall aspect of AutoCAD hasn't changed so much allowing users to easily migrate from one platform to another.

From the AutoCAD Plugin Developer's perspective, it is very important to understand those differences and how they could potentially affect their applications. More than that, it is very important to take advantages of each Platform's exclusive features and use them to empower your applications so they also feel very fluid and native to these Platforms.

Even by having shared **C++** source code the Mac OSX plugin will require new languages and concepts to be considered like **Objective-C** and **Cocoa**. This will add some new challenges to developers because those new languages have different syntax and requirements that Developers will need to understand, support and conciliate with Windows specific source code.

Requirements

We will use into this class the version 2016 for both Windows and Mac. Inside Windows this will require Developers to use **Visual Studio 2012**. Inside Mac OSX, even the native AutoCAD Mac code was created with Xcode 5, we will use **Xcode 6.4**. It is possible to develop and maintain shared Platform applications working on two different computers running the specific system but it is extremely painful. It is strongly recommended to adopt one system capable of emulate the two systems into a single machine.

Mac OSX does require specific Apple Hardware to run (not considering unsupported system hacks floating around but I'm pretty sure you won't add this complication to your daily work). Even being a limitation you can run Windows inside a Mac OSX system using Virtual Machine solutions (I would recommend either **Parallels Desktop** or **VMWare Fusion** which are the two most used and reliable). By doing that you will be able to run both Operating Systems into a single machine thus making your life much easier.

Shared C++ core

The idea of a shared C++ core to be consumed and used by both Platform versions of any applications is subjected to some requirements like the Platform Architecture (**32/64-bit**), supported languages and compilers. It is possible to share compiled static libraries but here we will focus on shared source code suitable to be compiled into Windows through **Visual Studio** and into Mac OSX through **Xcode**.

If you are planning to port your existing application to AutoCAD for Mac and also keep it going into the Windows System, some intensive work may be necessary to separate and reorganize your code accordingly. If you bind your interface to the code logic, not using appropriate **Software Design Patterns** like **MVC**, you will probably face some issues. In other hand, if it is a brand new code, you will be just fine.

Inside Xcode, the CPP files are well recognized and compiled but to be able to connect the pure C++ code (and **ObjectARX SDK**) to specific Mac OSX APIs like **Cocoa**, you will need to do it via **.MM** files which are files where C++ code can live with **Objective-C/Cocoa**. Further, Xcode UI design reinforcing the use of **MVC Design Pattern** so your interfaces will need to follow this pattern. Here is where a new challenge arises, how to share an application logic layer and bing it to two completely different UI strategies.

Here is a brief AutoCAD and Platform API comparison:

API / Library	Windows	Mac OSX
ObjectARX C++	Ok	Ok
Objective-C / Cocoa	Not supported	Ok
Win32	Ok	Partial
MFC	Ok	Limited support
ActiveX / COM	Ok	Not supported
.NET	Ok	Not supported
LISP	Ok	Ok
DCL	Ok	Not supported
Qt	Ok	Ok
Javascript	Ok	Not yet

Polymorphic Types

Once you plan to develop applications to be supported and executed by different platforms you need to be aware of those platform differences. The use of integer numbers, being 8,16 or 32-bit and also the use of the long type, are subjected to the Architecture of each platform. Autodesk does provide the Polymorphic types exactly to support this scenario so a number saved as a long integer into Windows will properly work into the Mac. The polymorphic types can be accessed through the "**adesk.h**" include file available on both Mac and Windows.

Unlike Windows-64, on OSX-64, a long is 64-bits. Any code that assumes **sizeof(int) == sizeof(long)** will potentially have bugs. The fix is to consistently use **Adesk::Int32** and **Adesk::UInt32** in place of longs so we can normalize this all between the two platforms. Remember that the storage size of number will depend on their declaration so if you plan to store a number that varies from 0 to 100, **Adesk::Int32** is too big for that and you will waste storage space.

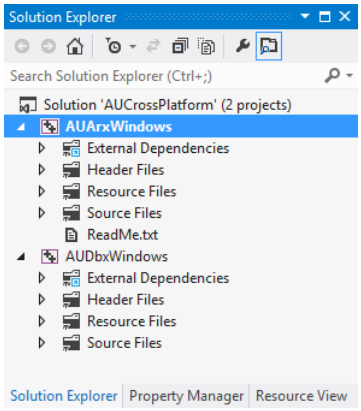
The polymorphic types are compiled accordingly to the Platform Definition ***_ADESK_MAC_*** and ***_ADESK_WINDOWS_***. You should also use these two definitions to make portions of your code suitable to the specific platform.

Creating Windows Project

Our first step will be the shared core module. This module will be a simple DBX module that will contain a custom entity. We will create the project inside Visual Studio 2012 with ObjectARX 2016.

Here are the requirements:

- Make sure you have **AutoCAD 2016** installed;
- Make sure you have **ObjectARX 2016** also installed;
- Use **Visual Studio 2012**;



We will create a blank solution called "***AUCrossPlatform***" where we will add both the DBX and ARX projects. The DBX project will be named as "***AUDbxWindows***" and the ARX project named as "***AUArxWindows***".

The projects were created using Autodesk **ObjectARXWizards 2016** available here:

http://images.autodesk.com/adsk/files/ObjectARXWizards-2016.zip

Configuring the Solution

Note that for each Platform build configuration (**Win32** and **x64**) you have the appropriate files referenced. This will ensure the necessary include files and library files are found. The ObjectARX SDK has specific include and binary files for both.

With both ARX and DBX projects created, now we have to configure the appropriate project dependency between them. You can do this by right clicking at the **AUArxWindows** project node, select the "***Project Dependencies…***" option. At the dependencies dialog, select the **AUDbxWindows** project from the list.

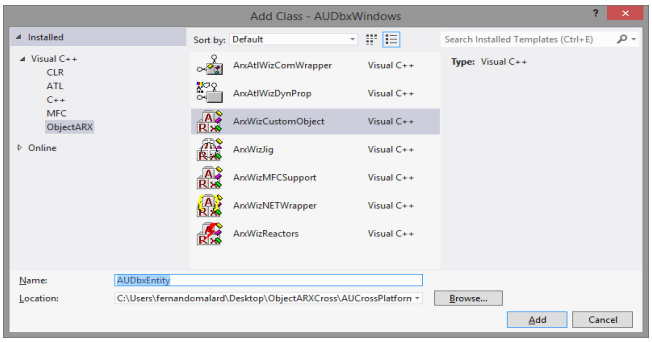
Once you properly create the dependency it is time to add the DBX library reference into the ARX project so they can properly link. This will allow the ARX module to consume the classes defined into the DBX project at runtime. Go to the ARX Project Settings; select at the top dialog combo boxes "***All Configurations***" and "***All Platforms***". Then go to ***Configuration Properties > Linker > Input > Additional Dependencies*** and add the following reference:

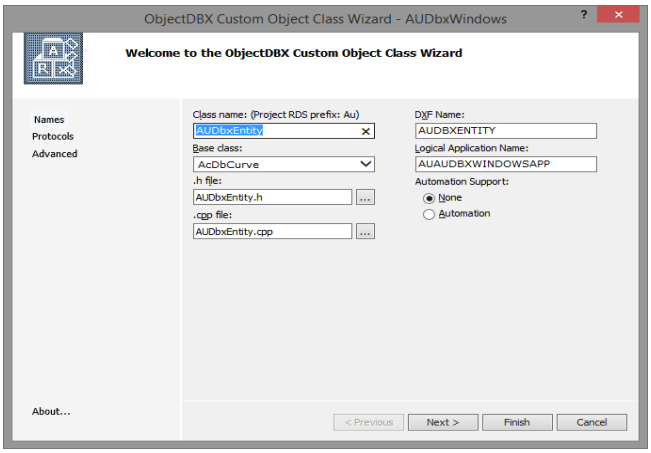
$(SolutionDir)\AUDbxWindows\$(PlatformTarget)\$(Configuration)*.lib

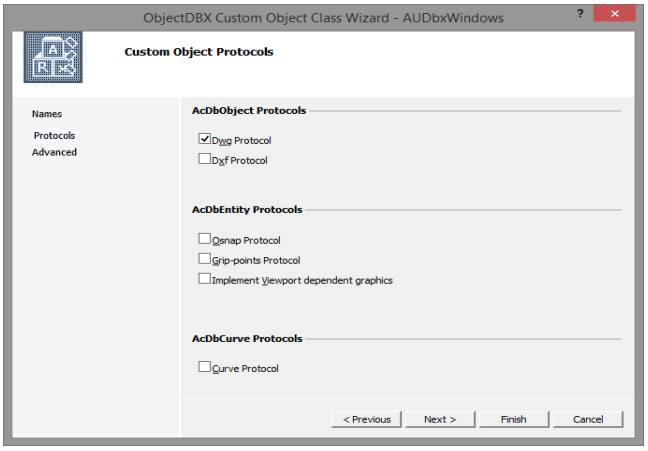
Creating the DBX Custom Entity

Now that we have our projects configured it is time to create a simple custom entity to be used for testing. We will use the ObjectARX ClassWizard to create the class. To do that, right click the **AUDbxWindows** project, go to "**Add >**" and select the "**Class...**" option. Once inside the "***Add Class***" dialog, select "***ObjectARX***" in the "***Visual C++***" node and then select "***ArxWizCustomObject***" at the list. Fill the "***Name***" with "**AUDbxEntity**". Click **Add**.

At the new dialog, fill the class name with "**AUDbxEntity**" and select "**AcDbCurve**" as the base class. Click **Next** and accept the defaults with "***Dwg Protocol***" checked. Click **Next** again, nothing checked and then **Finish**. The following 3 images illustrate the ObjectARX Class Wizard steps:







Next, we will declare the necessary class members at the **AUDbxEntity.h** file:

```
Adesk::Int16  _iTemperature;
AcString      _sText;
AcGePoint3d   _pts[2];
```

And we will create their access methods:

```
void SetTemperature(Adesk::Int16 iTemperature);
Adesk::Int16 GetTemperature() const;

void SetText(const ACHAR* sText);
const ACHAR* GetText() const;

void SetStartPoint(AcGePoint3d pt);
AcGePoint3d GetStartPoint() const;

void SetEndPoint(AcGePoint3d pt);
AcGePoint3d GetEndPoint() const;
```

Basically each method will set or retrieve the member data. Note that we need to use the proper assert method so AutoCAD will take care of file paging process when those data members change (**assertReadEnabled**() / **assertWriteEnabled**()). We will also need to page In/Out the date through the dwg filer methods:

```
// AUDbxEntity::dwgOutFields
pFiler->writeInt16(_iTemperature);
pFiler->writeString(_sText);
pFiler->writePoint3d(_pts[0]);
pFiler->writePoint3d(_pts[1]);

// AUDbxEntity::dwgInFields
pFiler->readInt16(&_iTemperature);
pFiler->readString(_sText);
pFiler->readPoint3d(&_pts[0]);
pFiler->readPoint3d(&_pts[1]);
```

To be able to see our custom entity at the screen we will implement a simple drawing routine inside **subViewportDraw**() method but first we need to return **Adesk::kFalse** from the **subWorldDraw**() method so the viewport version get called:

```
Adesk::Boolean AUDbxEntity::subWorldDraw (AcGiWorldDraw *mode) {
	assertReadEnabled () ;
	//------ Returning Adesk::kFalse here will force viewportDraw() call
	return (Adesk::kFalse) ;
}
```

Next, we can create the drawing routines. Basically we will trace a simple straight line from start point to the end point. Using this direction, we will then draw the Text, aligned with this line, containing the **_sText** content and the **_iTemperature** value. The line will be drawn using color 1 (**RED**) and the text color 3 (**GREEN**). This can be achieved by setting the color to the subEntityTraits() which is responsible for configuring the current drawing graphics:

```
void AUDbxEntity::subViewportDraw (AcGiViewportDraw *mode) {
	assertReadEnabled () ;
```

```
        AcDbPolyline plBuff;
        plBuff.addVertexAt(0,_pts[0].convert2d(AcGePlane::kXYPlane));
        plBuff.addVertexAt(1,_pts[1].convert2d(AcGePlane::kXYPlane));
        mode->subEntityTraits().setColor(1);
        mode->geometry().pline(plBuff);

        AcString strMsg;
        strMsg.format(_T("%s > %d"),_sText.constPtr(),_iTemperature);
        AcGeVector3d vDir = _pts[1] - _pts[0];
        AcGeVector3d vecPerp = vDir.perpVector().normalize();
        double dLen = vDir.length();

        mode->subEntityTraits().setColor(3);
        mode->geometry().text(_pts[0] + vecPerp*(dLen / 20.0),
           AcGeVector3d::kZAxis,vDir,(dLen / 10.0),1.0,0,strMsg);
}
```

Adding the Custom Entity to Model Space

The recipe for adding a new entity to the Model Space is quite simple. We need to open the Model Space for write, instantiate our entity through a pointer, set its properties and add it to Model Space closing the entity's pointer at the end. We will create a utility method inside the **acrxEntryPoint.cpp** file (project **AUArxWindows**) at the **CAUArxWindowsApp** class:

```
// Entity util
static void CreateCustomEntity(ads_point pti, ads_point ptj, ACHAR* pText, int iTemp)
{
        AUDbxEntity* pEntity = new AUDbxEntity();

        pEntity->SetStartPoint(asPnt3d(pti));
        pEntity->SetEndPoint(asPnt3d(ptj));
        pEntity->SetText(pText);
        pEntity->SetTemperature(iTemp);

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

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

        AcDbObjectId entId;
        pBlockTableRecord->appendAcDbEntity(entId, pEntity);
        pEntity->close();
        pBlockTableRecord->close();
}
```

The **AUArxWindows** project won't recognize the custom entity class unless we add its header file, which is located inside the **AUDbxWindows**:

```
#include "..\AUDbxWindows\AUDbxEntity.h"
```

Creating basic command prompt interaction

To allow the user to interact with our application and create an instance of the DBX custom entity, we will need to provide a command registered at AutoCAD command stack and perform some basic calls to aced…() command methods:

```
// Prompt version
static void AuMyGroupAUCROSSPROMPT () {

	ads_point pti, ptj;
	if (acedGetPoint(NULL, _T("\nClick at the start point:"), pti) != RTNORM)
		return;

	if (acedGetPoint(pti, _T("\nClick at the end point:"), ptj) != RTNORM)
		return;

	ACHAR pText[255] = _T("");
	if (acedGetString(1,_T("\nEnter the Text Message:"),pText) != RTNORM)
		return;

	int iTemp = 0;
	if (acedGetInt(_T("\nEnter the Temperature:"),&iTemp) != RTNORM)
		return;

	CreateCustomEntity(pti, ptj, pText, iTemp);
}
```

Further, this command needs to be registered at the end of **acrxEntryPoint.cpp** file:

```
IMPLEMENT_ARX_ENTRYPOINT(CAUArxWindowsApp)

ACED_ARXCOMMAND_ENTRY_AUTO(CAUArxWindowsApp, AuMyGroup, AUCROSSPROMPT, AUCROSSPROMPT,
ACRX_CMD_MODAL, NULL)
```

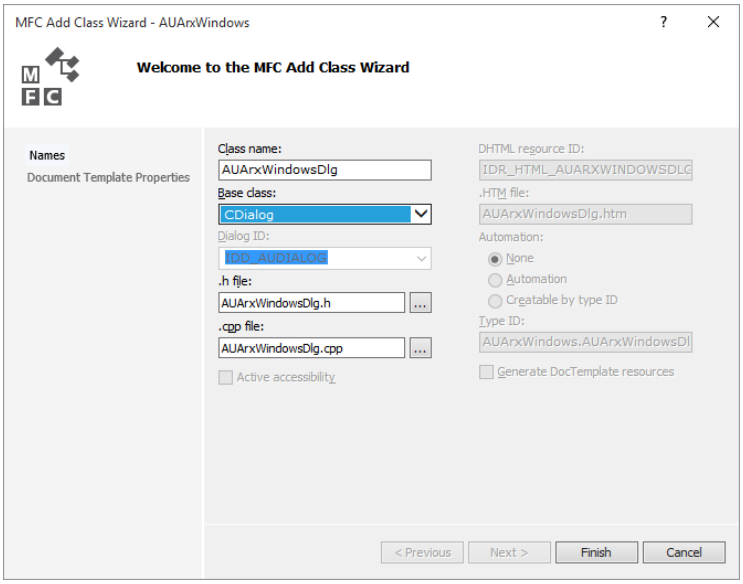
Creating the Windows UI with MFC

Now that we have our prompt version, let's create a **MFC** dialog so we can provide some UI capabilities to our **AUArxWindows** module. A separate file will handle the MFC dialog so we can skip its inclusion inside the Mac OSX project where we will implement another UI using OSX specific technology called **Cocoa**.

The dialog will be simple and contain just two fields to input the custom entity's text and temperature. They will be both an Edit control and will return a string value when the dialog closes. After the dialog is closed the user will then specify the start and end points at the AutoCAD screen.

To add a new **MFC** dialog to our **AUArxWindows** project we need to first create its Resource by accessing the Project Resource View. Once there, you can right click the "**AUArxWindows.rc**" node and select **Add Resource…** to display the Resource type selection dialog. Inside this dialog, simply select the Dialog item and then click the **New** button.

A new blank dialog will be created with default **OK** and **Cancel** buttons. It does have the default identification named as "**IDD_DIALOG1**". We will change this to "**IDD_AUDIALOG**". Now, right click at the dialog center screen and select "**Add Class...**". The MFC Class Wizard dialog will appear. Change the class name to "**AUArxWindowsDlg**" and the base class to "**CDialog**". The suggested .h and .cpp file names will be updated accordingly. Click **Finish**.

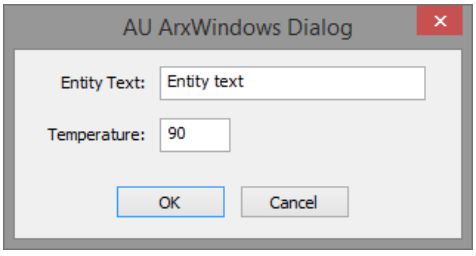


Now, go to the just created "**AUArxWindowsDlg.h**" file and add the include for the resource file:

```
// AUArxWindowsDlg dialog
#include "Resource.h"
```

The dialog is created and now we need to add two edit boxes to receive the user input before creating our custom entity into Model Space. After adding the Edit controls, add a static text in front of each one. Adjust the visual layout to controls get aligned and reduce the dialog size accordingly. You can keep both the **OK** and **Cancel** buttons. Change the name of your dialog as you wish. All these operations are done via Visual Studio **Properties** palette and the controls can be inserted via **Toolbox** palette.

It is recommended to change the Edit box identifications, we will use **IDC_EDIT_TEXT** for the text input box and **IDC_EDIT_TEMP** for the temperature input box. The latter we will also change its property "**Number**" to **True**.



Now right click at each Edit control and select "Add Variable…" naming them as "**edtText**" and "**edtTemp**". Next, double click at both OK and Cancel buttons so you generate a method callback for them.

Further, add some other local variables to carry the values out of the dialog. We will need to include the dialog header file into the **acrxEntryPoint.cpp** file:

```
#include "AUArxWindowsDlg.h"
```

The new command will open the dialog then retrieve the values back:

```
// Dialog version
static void AuMyGroupAUCROSSDLG() {
	AUArxWindowsDlg dlg(CWnd::FromHandle(adsw_acadMainWnd()));

	if (dlg.DoModal() == IDOK)
	{
		ads_point pti, ptj;
		if (acedGetPoint(NULL, _T("\nClick at the start point:"), pti) != RTNORM)
			return;

		if (acedGetPoint(pti, _T("\nClick at the end point:"), ptj) != RTNORM)
			return;

		CreateCustomEntity(pti, ptj, dlg._sText.GetBuffer(), dlg._iTemp);
	}
}
```

And here is the command new macro to be added to the end of acrxEntryPoint.cpp file:

```
ACED_ARXCOMMAND_ENTRY_AUTO(CAUArxWindowsApp, AuMyGroup, AUCROSSDLG, AUCROSSDLG,
ACRX_CMD_MODAL, NULL)
```

This completes the Windows portion of our Cross Platform application so from now on we will deal with the Mac OSX side of the application. There we will reuse the non-UI portion of our Windows code (both DBX and ARX) and we will create a new UI to replicate the MFC dialog we have inside Windows.

Creating MacOSX Project

To be able to compile and build the same application and make it work with AutoCAD 2016 for Mac OSX we will need to use **Xcode** compiler. AutoCAD 2016 is built with Xcode 5, base SDK 10.9 and deploy target is 10.9. We will use **Xcode 6.4** for this sample project. The ObjectARX libraries are compatible with Xcode 6.4 so we won't have any problems.

AutoCAD used to support Mac OS in the past and a relatively big portion of AutoCAD source code is **C/C++** friendly being compatible with Mac OS. Other than that, Autodesk had a hard work adapting the latest AutoCAD versions to Mac OSX beginning with natural platform differences (like the lack of **Windows API**) and some additional challenges like features not portable at all (like **COM** interfaces). To make the job easier to Developers coming from Windows, Autodesk created a good amount of C++ header support files thus allowing users to reuse great part of their source code. This is true even for complex Windows classes like **CString**.

As a compiler and IDE, Xcode is quite different from Visual Studio and it has some advantages and limitations when we compare both side by side. Here is a brief comparison of their essential features:

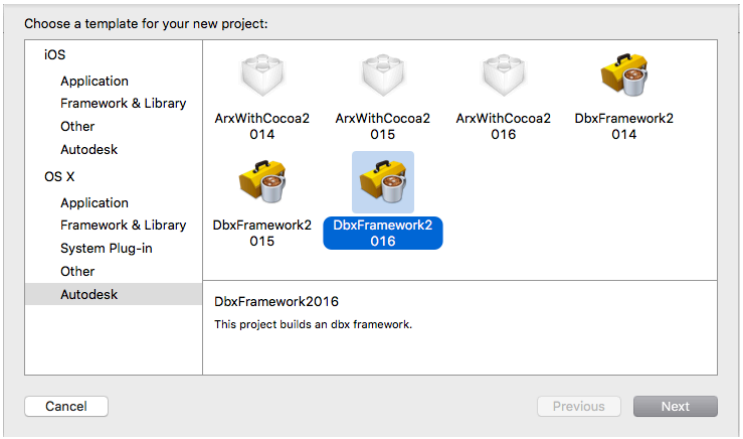
Feature	Visual Studio	Xcode
Integrated Environment	Solution (.sln)	Workspace (.xcworkspace)
Project	VC++ Project (.vcxproj)	Xcode Project (.xcodeproj)
Configuration	Solution Configuration	Schemes
Output Files	.arx / .dbx / .crx	.bundle / .dbx
Project Configuration File	.props	.xcconfig
Platform support	Win32 / x64	x86_64
Resources / Dialogs	.rc / MFC	.xib / Cocoa
C++ source file	.CPP	.CPP / .MM
Resource editor	Resource View	Interface Builder
Application Design Pattern	any pattern	MVC by default

To properly port our existing Windows code and complete it with Xcode specific files we will need to first set a shared folder (seen by either Mac OSX or Windows) where both source code files will coexist. Note that Autodesk does provide the ***_ADESK_MAC_*** definition, which will help us to define portions of our code to compile or not, depending on which Compiler is consuming it. For some compiler specific files, we don't need to even include them into the other compiler thus avoiding unnecessary processing or code detour via ***#ifdef*** instructions. This is the case of MFC dialog files in Visual Studio and Cocoa files in Xcode.

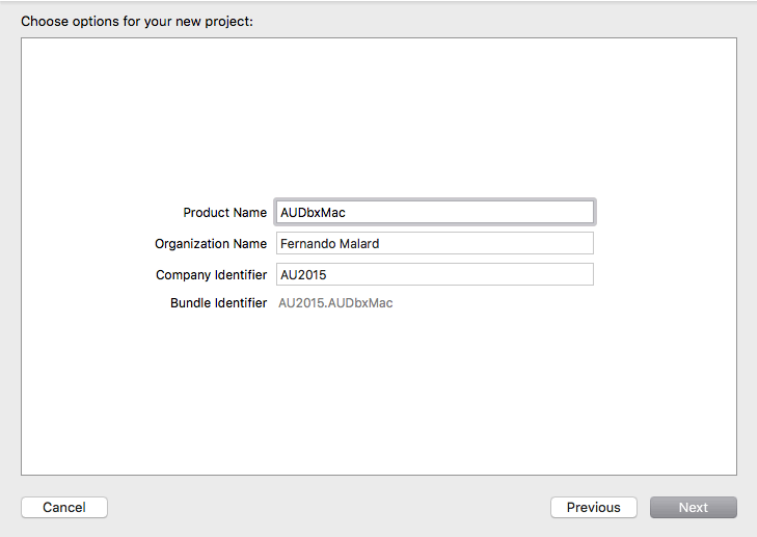
Each compiler will also require different project settings to accommodate your deployment strategy and multiple **ObjectARX** version support. Note that AutoCAD for Mac requires a **x64** machine so there isn't support to **32-bit** code thus reducing our multiple project settings in Xcode by a half.

Creating the Xcode projects

We will first create a blank Workspace into Xcode to accommodate our two projects. Open Xcode, ignore the Welcome screen, go to **File > New > Workspace…** then locate your Visual Studio Solution folder and save the Workspace as "**AUCrossPlatform.xcworkspace**". Within this blank Workspace, got to File > New and select **New Project…** to open the template selection dialog. At the left, under **OSX**, select **Autodesk** then choose "**DbxFramework"** and click **Next**. Note that you need to install ObjectARX for Mac SDK to be able to see the templates:

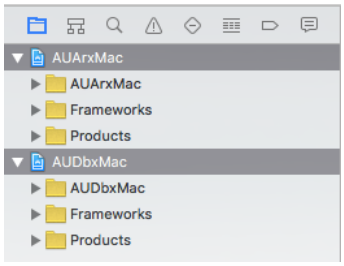


Name the Product as "**AUDbxMac**" and fill the other fields as you wish. Click **Next**.



It will suggest the root folder of your recently created blank Workspace. Accept that and click **Create** to finish. A new folder with your product name will be created and also a folder called "**Build**" will be created at your Workspace's root folder. This will be the build output folder.

Now, create the second Project into the same Workspace by right clicking at the blank left space. This time selects the "**ArxWithCocoa**" template, which is the ARX corresponding project type. Click **Next** and name it as "**AUArxMac"**. Click **Next** again and then click **Create** (it will default again using the Workspace's root folder to save the project). You should end up with a Workspace like this:



Install Path

Now we need to change one important thing into the **xcconfig** files. Dynamic Libraries are resolved inside **Mac OSX** in a different way when compared to Windows. Each dependency receives a path to whatever it links too. As we will need to link our Bundle (**ARX**) with the **DBX** to import our custom entity we need to assure the ARX module can find the DBX at load time. If you just let the Xcode to compile with the **default configuration**, the resulting link path will be:

$(LOCAL_LIBRARY_DIR)/Frameworks which resolves to **/Library/Frameworks**

But, it will require you to deploy a copy of your module into this folder every time you build it and this would be very annoying (not to mention the impact of debugging the application). To avoid that, we will

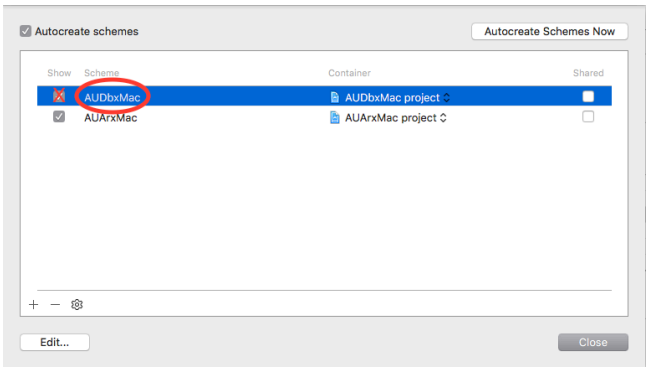
use the special install path called "**@rpath**". Starting in **Mac OSX 10.5**, this special path, when placed in front of an install name, will tell the dynamic linker to search a list of locations for the library. That list is embedded into the application, and can therefore be controlled by the application's build process, not by the framework. To fix this, simply open the "**dbx_common.xcconfig**" file into the **AUDbxMac** project and add this code at the end:

```
INSTALL_PATH = @rpath
```

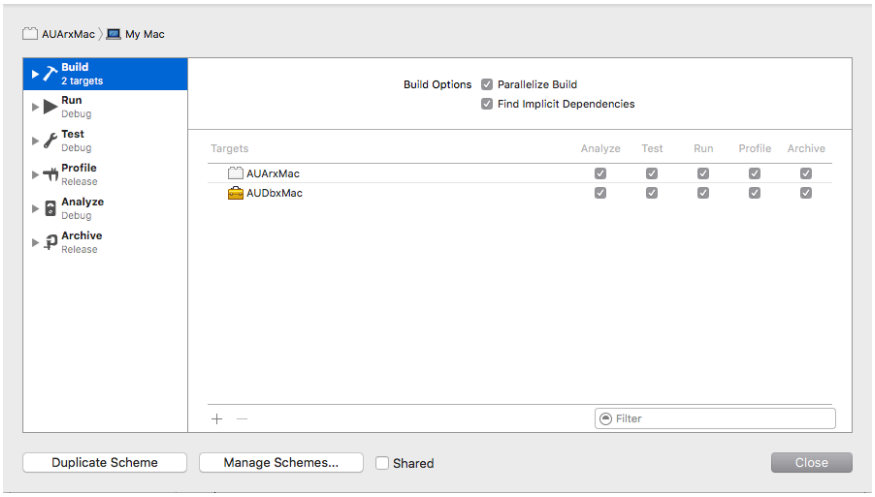
This will predefine the Installation Directory to be resolved as mentioned above. Repeat the modification into the "**arx_common.xcconfig**" file into the **AUArxMac** project.

Adjusting the Schemes and Debugging the Project

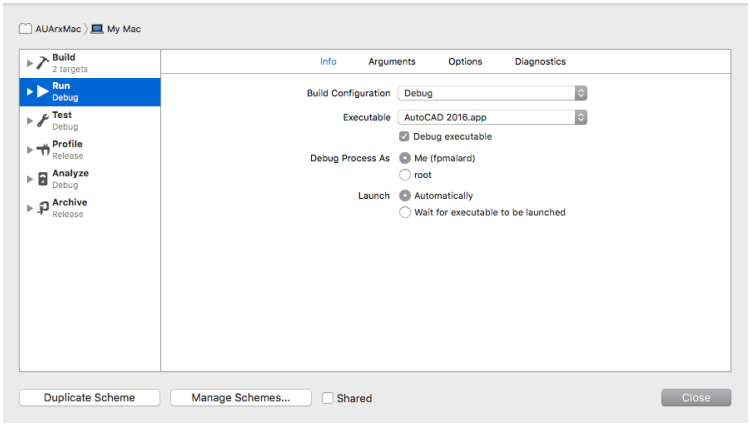
Now we will configure our Schemes so we can have a better environment to build and debug our code on AutoCAD 2016. We will keep only 1 Scheme which will compile the project **AUArxMac** (which will automatically compile the **DBX** module as we did a static reference to that project).



Select the **AUDbxMac** scheme, click at the "**-**" button, then select **Delete**. Now select the **AUArxMac** scheme, click at the **Edit…** button to enter into the **Scheme Editor** dialog. Select **Build** step on the left panel and, under **Targets** area, add the **AUDbxMac** target to the list so both get compiled when you build this Scheme.



After that, go to the **Run** step to specify the Application as the **Executable**, in this case, "**AutoCAD 2016.app**". Now you should be ready to build all the Schemes accordingly.

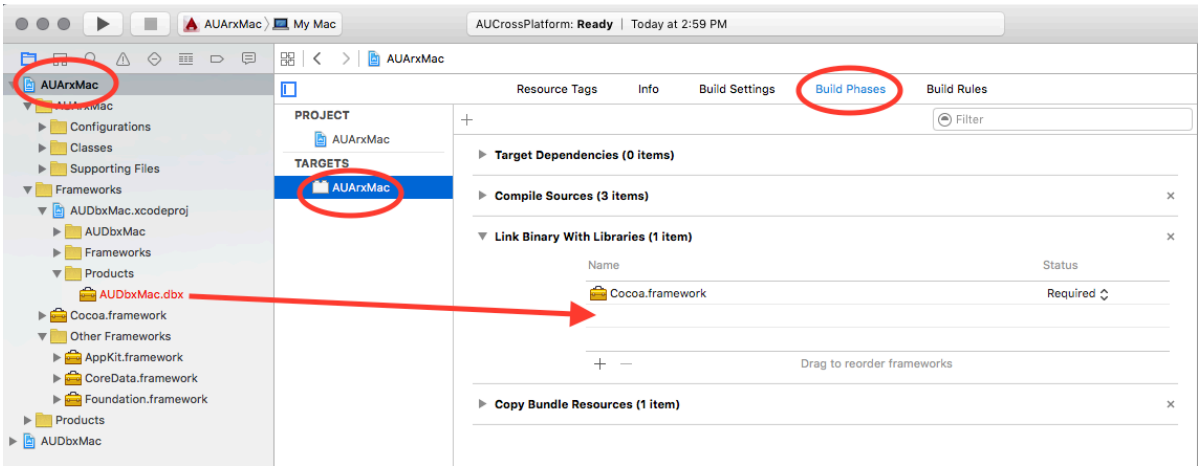


Adding the shared Windows files

Now we will add the files created by the Visual Studio Projects we want to share. We will start with the **AUDbxMac** project. Remove (and delete) the "**dbxmain.cpp**" file from the "**Classes**" folder. Now, add to the same folder, the existing files "***AUDbxEntity.h/.cpp***", "***acrxEntryPoint.cpp***" and "***StdAfx.h/.cpp***". Make sure you check both Targets at the bottom of Xcode **Add Files…** dialog.

Now go to the **AUArxMac** project, remove and delete the "**arxmain.cpp**" file, and add the files: "***acrxEntryPoint.cpp***", "**DocData.h/.cpp**" and "***StdAfx.h/.cpp***". Don't add the MFC Dialog files because we will create a new interface for them. Now, right click the "**Frameworks**" folder and add the **AuDbxMac** project file so our **AUArxMac** project creates a dependency to the DBX project where our custom entity is.

Once your projects are linked you can now drag the DBX module to the "**Link Binary With Libraries**" area thus making the ARX module to link with DBX:



Finally, we need to adjust the specific Windows code to be ignored by the Xcode. We will use the **_ADESK_MAC_** definition for that. Here is how the include code area should look like at the beginning of "**acrxEntryPoint.cpp**" file (the **extern** method will be used later):

```
#ifdef _ADESK_MAC_
    #include "../AUDbxWindows/AUDbxEntity.h"
    extern bool ShowAUArxCocoa(AcString& txtEntity, int& iTemp);
#else
    #include "resource.h"
    #include "..\AUDbxWindows\AUDbxEntity.h"
    #include "AUArxWindowsDlg.h"
#endif
```

In this file, add the **_ADESK_MAC_** inside the method **AuMyGroupAUCROSSDLG**() around the code to show the MFC dialog because we will ignore it when compiling for Mac OSX. Do the same around the **ACED_ARXCOMMAND_ENTRY_AUTO**() macro call for this command.

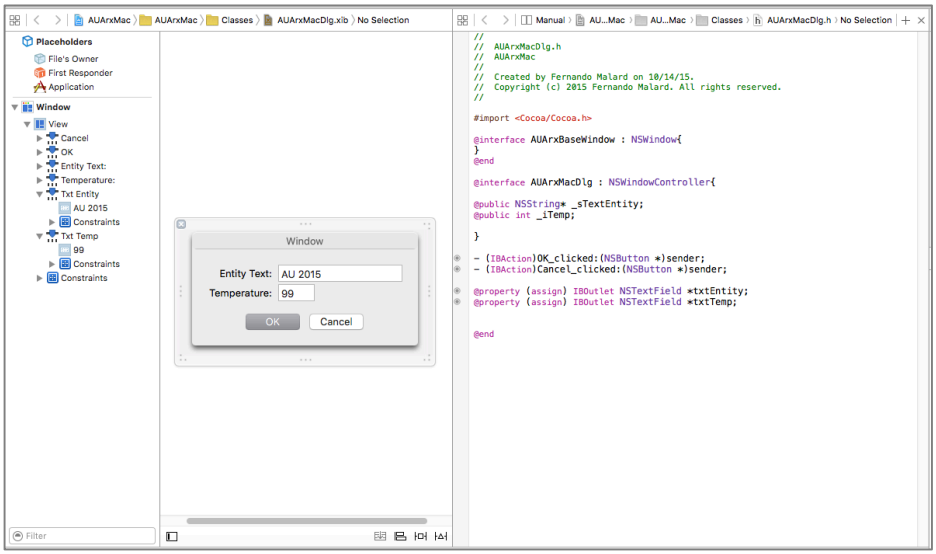
You can now **Build** the **AUArxMac** project which will first build the **AUDbxMac** project. The 2 modules will be generated and linked. Go ahead and open **AutoCAD 2016**, load first the DBX module and then the Bundle. Run the prompt version command and create the custom entity.

Creating the Mac UI with Cocoa and Objective-C

As we said before, Mac OSX UI is driven by Cocoa programming language so we will use it to reproduce the same dialog interface we created using MFC at the Windows portion of our Project. The first thing to do is to create the resource file. To do that, right click at "**Classes**" folder in **AUArxMac** project, select **New File…** and click at the "**Cocoa Class**" option under OSX / Source node at the left panel. Select the "**Objective-C**" language option, name the class as "**AUArxMacDlg**" subclass of "**NSWindowController**". Check the "**Also create XIB file for user interface**" option so it creates the corresponding resource file. Click **Next** and confirm the Target is checked at the bottom of this dialog. Click **Create**. Three new files will be created into your project: "**AUArxMacDlg.xib**, **AUArxMacDlg.h** and **AUArxMacDlg.m**".

Click on "**AUArxMacDlg.xib**" file, a blank Window will appear. Now, we will add the necessary controls trying to mimic what we did in Visual Studio. First, add two buttons to the dialog, one "**OK**" and another "**Cancel**". They can be found at the Xcode right panel as "**Push Button**" item. Rename them through "**Attributes Inspector**" tab (right panel). Now add 2 "**Label**" and 2 "**Text Field**" controls. Adjust the Layout and resize the window accordingly. Try to configure its visual as close as possible to what you did in Visual Studio.

Now we will connect the Buttons and Edit fields to the Windows Controller class. To do that, click at the **XIB** file and then, at the top right **Xcode** icon list, select the second ("**Assistant Editor**"). Once you do that the central panel will split in two vertical panels with the **XIB** at the left and its corresponding **Windows Controller** .h file (in this case, the file "**AUArxMacDlg.h**"). To insert the outlets / actions for the Buttons, press and hold the **CTRL key**, click and hold the **OK** button and start to drag it to the right within the **"@interface**" declaration.



When you see the "**Insert Outlet or Action**" label, release the button and a small dialog will appear. Select "**Action**" as the connection type, **NSButton** as the Type and name it as "**OK_clicked**". Click **Connect** and now you should have the action callback declaration added to your controller class. Repeat the process to the **Cancel** button:

```
- (IBAction)OK_clicked:(NSButton *)sender;
- (IBAction)Cancel_clicked:(NSButton *)sender;
```

Use the same technique to generate the "**Outlet**" for the the **TextField** controls:

```
@property (assign) IBOutlet NSTextField *txtEntity;
@property (assign) IBOutlet NSTextField *txtTemp;
```

Build the **AUArxMac** project to check if everything is ok. Next, we will add the code to display the dialog from the ARX command callback. As the ARX command callback is defined into a C++ code file we will need to change the dialog definition file type. It was created as "**Objective-C Source**" with the "**.m**" extension. We will need to change it to "**Objective-C++ Source**" by changing the "**.mm**" extension. To do that, simply select the "**AuArxMacDlg.m**" file click it so the rename in-place editor turns on, and change the file extension to "**.mm**".

We need now to change the default **NSWindow** base class by a simple custom class derived from NSWindow to override the **canBecomeMainWindow** method to return **NO**:

AUArxMacDlg.h:

```
@interface AUArxBaseWindow : NSWindow{
}
@end
```

AUArxMacDlg.mm:

```
// Base Window class
@implementation AUArxBaseWindow
- (BOOL)canBecomeMainWindow
{
    return NO;
}
@end
```

Further, we will add 2 placeholder variables to our dialog class. Change the **@interface** of **AUArxMacDlg** as follows:

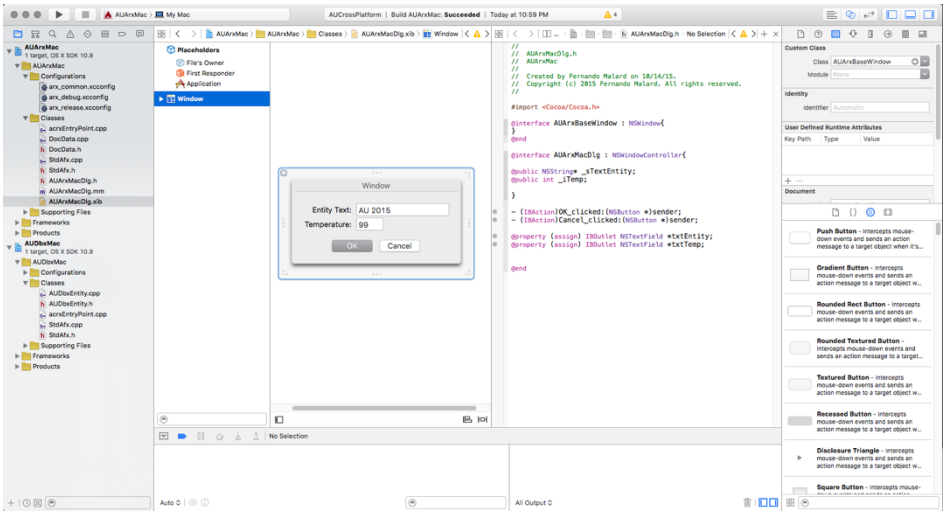
```
@interface AUArxMacDlg : NSWindowController{

@public NSString* _sTextEntity;
@public int _iTemp;

}
```

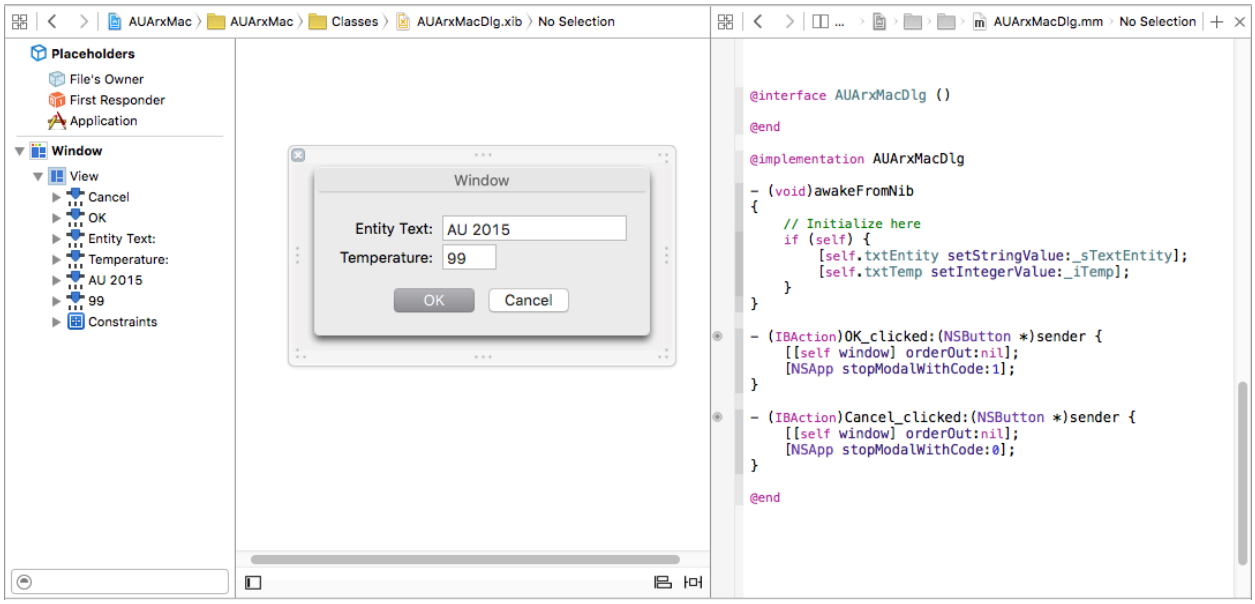
Now, select the **XIB** file, click at the **Window** component, go to the "**Identity Inspector**" tab, under "**Custom Class**", select "**AUArxBaseWindow**" from the drop down:

Go back to the **AUArxMacDlg.mm** file, add the method **awakeFromNib**() that will allow us to initialize the dialog controls. At the button events, we will just close the dialog with **1 for OK** and **0 for Cancel**.



The Window caller will then be able to figure out how the user did finish the dialog:

```
- (void)awakeFromNib
{
    // Initialize here
    if (self) {
        [self.txtEntity setStringValue:_sTextEntity];
        [self.txtTemp setIntegerValue:_iTemp];
    }
}
- (IBAction)OK_clicked:(NSButton *)sender {
    [[self window] orderOut:nil];    // hides it
    [NSApp stopModalWithCode:1];
}
- (IBAction)Cancel_clicked:(NSButton *)sender {
    [[self window] orderOut:nil];    // hides it
    [NSApp stopModalWithCode:0];
}
```



Now, we will add a **C++ function** into this **.mm** file to do the bridge between the **Objective-C language and C++**. This function will allocate the Window, initialize the local variables and then run this dialog as a Modal Window returning **0** or **1** value.

```
#include "AcString.h"

bool ShowAUArxCocoa(AcString& txtEntity, int& iTemp)
{
    AUArxMacDlg* pWnd = [AUArxMacDlg alloc];
    const wchar_t* pText = txtEntity;

    pWnd->_sTextEntity = [[[NSString alloc] initWithBytes:pText
       length:wcslen(pText)*sizeof(wchar_t)
       encoding:NSUTF32LittleEndianStringEncoding] autorelease];
    pWnd->_iTemp = iTemp;

    [pWnd initWithWindowNibName:@"AUArxMacDlg"];
    int iRes = [NSApp runModalForWindow:pWnd.window];

    iTemp = [pWnd txtTemp].intValue;

    NSData* d = [[pWnd txtEntity].stringValue
             dataUsingEncoding:NSUTF32LittleEndianStringEncoding];
    std::wstring sRet = std::wstring((wchar_t *)[d bytes],
       [d length]/sizeof(wchar_t));
    txtEntity = sRet.c_str();

    [pWnd release];
    return (iRes > 0);
}
```

The last step is now change the **AuMyGroupAUCROSSDLG**() method so we can open the **MFC** dialog when running in Windows and the **Cocoa** window when running in Mac OSX. Here is the modified method code:

```
static void AuMyGroupAUCROSSDLG()
{
    bool bCreate = false;
    AcString txtEntity = _T("Autodesk");
    int iTemp = 14;

     #ifdef _ADESK_MAC_
        bCreate = ShowAUArxCocoa(txtEntity, iTemp);
     #else
        AUArxWindowsDlg dlg(CWnd::FromHandle(adsw_acadMainWnd()));
        bCreate = (dlg.DoModal() == IDOK);
        txtEntity.format(_T("%s"),dlg._sText.GetBuffer());
        iTemp = dlg._iTemp;
     #endif

    if (bCreate)
    {
        ads_point pti, ptj;
        if (acedGetPoint(NULL, _T("\nClick at the start point:"), pti) != RTNORM)
            return;
        if (acedGetPoint(pti, _T("\nClick at the end point:"), ptj) != RTNORM)
            return;

        CreateCustomEntity(pti, ptj, (ACHAR*)txtEntity.constPtr(), iTemp);
    }
    else
        acutPrintf(_T("\nDialog cancelled."));
}
```

As a final touch, select the **XIB** file at the Project browser, go to the Attributes Inspector tab, under Controls, uncheck the **Close, Resize and Minimize** options. Build the Workspace again and perform a new test.

Conclusion

There is much more than what we presented here. A complex **Cross Platform** Plugin will demand many other actions regarding deployment, menus, installers and documentation. The main idea of this class was to provide an initial kick-off so you can at least understand all the challenges involved here. As the **AutoCAD for Mac** family increases and expand its user base (as the **Mac OSX** operating system itself) more and more users will demand for solutions targeting both Windows and Mac.

There isn't indeed one best and ultimate platform to work on. There is much more around the AutoCAD product itself that may cause impact and change the way you use your machine. We have a huge **Internet** wave hitting the shore with tons of mobile and **Cloud** solutions. This will add an extra level of complexity to our work.