Alaska Software ActiveX Tutorial

Tutorial: ActiveX with Xbase++

Based on a real world problem "Integrating Excel capabilities into your Xbase++ application" this Technote gives you first hand knowledge using ActiveX in the context of Xbase++. Focusing on practical aspects of ActiveX, this Technote tries to avoid going into the details of the underlying technologies and concepts related to ActiveX, such as ActiveX/OCX/COM/DCOM and OLE, just to name a few. Nevertheless, working through this Technote should put you in the position of being able to utilize capabilities provided by other ActiveX servers, such as MS Word by your own. As a prerequisite, only basic experience with Xbase++ and its object-oriented programming model is required.

Xbase++ ActiveX support

Alaska Software has released an early version of its ActiveX library for Xbase++ called XPPOCX. The latest version requires Xbase++ 1.8 and higher and provides an easy way of ActiveX integration. In fact, the XPPOCX library is some sort of sneak preview into the forthcoming Visual Xbase++ (VX) ActiveX support.

Currently, the Xbase++ ActiveX support library is available for free to all Subscription customers who have renewed their subscription. It was intended as a Thank You for the early birds of the Developer Subscription renewal program. The ActiveX support library comes with ready to use sample implementations of Xbase++ classes that utilize the following OCX controls:

OCX controls	
Internet Explorer	display any HTML content
MS Common Dialog	usage of a calendar class
MS RTF	integrate RTF documents

If you are inspired by this article feel free to download the "ActiveX support" from Alaska Software's ESD system (Electronic Software Delivery).

Here is how it works: Please send an email to esd@us.alaska-software.com, with the word "HELP" in the subject line. The ESD system will respond with an email containing instructions that guide you through the authentication process. Be sure to have your EO (Order ID #) handy which you received with your Subscription Renewal Confirmation.

Note: the source code included with this article requires Xbase++ 1.8 .

What Is ActiveX?

Have'nt you asked yourself why you seem to re-invent the wheel over and over again? Do I really have to write my own HTML rendering engine when IE, Netscape or any other browser already has one integrated? Do I really have to write my own word processor, or is there one which I can integrate into my application, one which my customers are already familiar with? If you asked yourself these or similar questions, you are the perfect audience for this article, because here, ActiveX could come into play.

ActiveX is a technology that allows you to take advantage of an existing application program and to incorporate its functionality into your own applications. COM (Component Object Model) is the standard architecture on which the technology is based and which an ActiveX component builds on to. A component can be viewed as an object composed of other objects. Each component exposes a set of interfaces through which all communication to the component is handled. Interfaces provide methods and properties. Access to methods and properties is called OLE automation - a great term for the simple fact of calling a method or getting data from a property alias instance variable.

The ActiveX component which provides the interface functionality is a "server" component, and the application which uses the functionality of an ActiveX component is the "client".

How to Find Out What ActiveX Controls are Installed and What Features They Provide

A Windows PC already has many ActiveX components installed, many of which are used as controls. So the terms Control and Component are used synonymously in this article. To investigate them, an ActiveX Object Viewer is needed but normally not installed on a Windows PC, which was the case with my PC. So, I had to find an ActiveX Object Viewer and went to http://www.yahoo.com where I searched for "ocx viewer". Yahoo returned many links, and this one looked the most promising:

```
http://www.microsoft.com/com/resources/oleview.asp
```

After reading through the brief description I decided to download it:

```
http://www.microsoft.com/com/resources/OVI386.EXE
```

A few minutes later the installation was complete and I had a utility which let me browse the ActiveX components on my PC. What a collection!

While I was scanning through the ActiveX components I found Excel, and quickly decided on a project to use MS Excel's capability to present a set of values as a chart and print it.

Again, I used the OLE Viewer to search for the term Excel. I searched the "All Objects" section and found "Microsoft Excel Application" (plus a few more described as Excel

related). Opening this tree brought up "Application", and after a double click, I came to "View Type Info".

Note: This feature - that an utility like OLE Viewer can provide information on an ActiveX component - simply works because every ActiveX component describes which functions, methods and properties it exposes. ActiveX components are "self descriptive", i.e. information on ActiveX components is stored in a so-called Type Library.

A type library is a file, or part of a file, that provides information about the interfaces of a COM object. More specifically, the type library contains information about classes. A class is a description of an object. Type libraries themselves do not store actual objects; they only store information about those objects.

A type library specifies all the information required by an automation client for calling an object's method or inspecting its properties. For properties, the type library describes the stored value. For methods, the type library provides a list of all the arguments the method can accept, tells you the data type of each argument, and indicates whether an argument is required or not.

An object by itself does nothing unless you tell it do something. Generally speaking, objects provide an interface to a certain feature or functionality to your application. To programmatically examine or control an object, you can use the properties and methods that the object supports. A property is a function that sets or retrieves an attribute for an object. A method is a function that performs some action on an object.

Let's Talk Xbase++

Examining the type library of the Excel component (EXCEL8.OLB or EXCEL9.OLB) with OLE Viewer displays the constants, properties and methods exposed by this ActiveX component. But what are these in Xbase++ terms?

As a rule of thumb you could think

OLE Viewer / ActiveX terms	**Xbase++ equivalents**
interface	class
object	instance
constants / enums	#defines
properties	iVars
methods	methods

Personally, I found OLE Viewer to be a good tool for getting an overview of which components are installed on your system, but it is not a very good tool to describe which interfaces, constants, properties and methods an ActiveX component exposes. So I again connected to the internet and found

```
http://www.devcomponents.com/comassistant/
```

and downloaded

```
http://www.devcomponents.com/downloads/comassistantsetup.exe
```

This neat utility extracts the type library information and creates a set of HTML files which give a complete description of the respective ActiveX control (I used EXCEL9.OLB) as an HTML help file. Great! Be careful, though, more than 50 MB of HTML documentation with > 10.000 HTML files is created from the Excel type library!

Side Note: HTML WorkShop has to be installed as well. If you do not have it yet, it may be installed from <AlaskaCD>:\HTMLHELP

Identifying the Task

The task I wanted to solve is: print a well designed pie chart right to the printer using Excel's functionality. This sounds quite simple, but I was neither familiar with the object model of Excel nor with its interfaces. So were should I start? Very easy, I had Excel tell me what to do via its Macro recorder. That's how it works:

Start MS Excel. On the Tools menu, click Macro, and then select "Record New Macro". In the "Store Macro In" drop-down box, select the name of the active document. Make a note of the new macro's name, and then click OK to start recording. Now perform all steps required to accomplish the task and stop the macro recorder.

On the Tools menu click Macro, and then select Macros. Choose the name of the new macro in the list and click Edit. Now the Visual Basic Editor displays the recorded macro as Visual Basic code. It contains all names for properties and methods and all that is left to do is to translate this code to Xbase++.

Beginning Our Project

To start a new project, select an empty folder and create a PRG file with a Main() function, which is the starting point. I labelled mine GRAPH.PRG, for instance. Then perform these commands:

```
DIR *.prg /b > Graph.txt
PBuild @Graph.txt
```

Edit the resulting PROJECT.XPJ and change the GUI switch to:

```
GUI  = yes
```

Now let's look at the VBA code that I generated with Excel's macro recorder. Here is the entire VBA script:

```
01: /*  VBA (Visual Basic for Applications)
02: Sub Macro1()
03:   '
04:   ' Macro1 Macro
05:   ' Macro recorded 04.08.2002 by Frank Grossheinrich
06:   '
07:   '
08:   ChDir "H:\Alaska\OCX\Tutorial2"
09:   Workbooks.Open Filename:="H:\Alaska\OCX\Tutorial\SALES.DBF"
10:   Charts.Add
11:   ActiveChart.ChartType = xl3DPieExploded
12:   ActiveChart.SetSourceData Source:=Sheets("SALES").Range("A2:B5"), _
13:      PlotBy:=xlColumns
14:   ActiveChart.Location Where:=xlLocationAsNewSheet, _
15:      Name:="PrintGraph"
16:   With ActiveChart
17:      .HasTitle = True
18:      .ChartTitle.Characters.Text = "Xbase++ ActiveX Sample"
19:   End With
20:   ActiveChart.HasLegend = True
21:   ActiveChart.Legend.Select
22:   Selection.Position = xlCorner
23:   ActiveChart.ApplyDataLabels Type:=xlDataLabelsShowValue, _
24:      LegendKey:=False,HasLeaderLines:=True
25:   ActiveWindow.SelectedSheets.PrintOut Copies:=1, Collate:=True
26:   ActiveWorkbook.Close
27: End Sub
28: */
```

I am going to translate this VBA code to Xbase^{++}, but first we need to prepare Excel and initialize its capability to act as an OLE automation server:

```
#pragma library("xppocx.lib")
#pragma library("xppui3.lib")

PROCEDURE Main
   oExcel := XbpOcx():new( AppDesktop(), "Excel.Application" )
   oExcel:create()

RETURN
```

This Xbase^{++} code creates the Excel application object from the XbpOcx() class and tells the linker which libraries to link with two #pragma instructions. That means, you must have these DLLs in your PATH:

```
XPPOCX.DLL
XOCMAIN.DLL
XPPUI3.DLL
```

The XbpOcx() class is not yet documented, but will be with a future version. For now we just need a few methods of the class, such as *:new()*, *:create()* and *:destroy()*, for the lifecycle of an ActiveX component, and three other methods, *:getProperty()*, *:setProperty()* and *:callMethod()*, which are the most important ones to set or retrieve properties of a component or to call a method of a component. The method names should be self explanatory.

XbpOcx()

Method	Parameters	Return
:new()	oParent, cCLSID, aPos, aSize	self
:create()	oParent, cCLSID, aPos, aSize	self
:destroy()		self
:getProperty()	cProperty	axValue
:setProperty()	cProperty, axValue	axValue
:callMethod()	cMethod, axParam, ...	axValue

The methods *:new()*, *:create()* and *:destroy()* have the same meaning as for Xbase Parts, only the second parameter differs: it is the Class-ID of the ActiveX component to be accessed. The Class-ID is ussually a cryptic Hex number that can be obtained from the OLE Viewer. Creating an application object is much easier using the application's name. So, "Excel.Application" can be used as Class-ID for the Excel application object.

The two methods *:setProperty()* and *:callMethod()* accept as first parameter a string with the name of the desired property or method. The second parameter *axValue* can be passed either by value or as an array.

The easiest and most common approach is passing parameters by value. This means that standard parameters such as strings, logicals and numerics can be passed as is. No further conversion is needed (bear in mind that an ActiveX component is written in C/C++ or another strong typed language and needs parameters to be passed as specific types which are integer, floats, handles, etc). That means, if a method requires a directory name as string you can pass the string as is. Example:

```
cInputTable := "c:\temp\sample.dbf"
oBook := oWorkbook:callMethod( "Open", cInputTable )
```

cInputTable is a standard Xbase^{++} string. The same applies to logical and numeric values if the method (:callMethod()) or property (::setproperty()) require these.

The other technique, using strong typing syntax, is to pass the parameter as an array containing two elements: a constant which specifies the type, and the data value. Example:

```
oBook := oWorkbook:callMethod( "Open", { VT_BSTR, cInputTable } )
```

This code has the same effect as the one above.

Note: These strong typing constants are defined in XBPOCX.CH

Because ActiveX routines use strong typing, the return values of *:setProperty()*, *:getProperty()* and *:callMethod()* are also in this array format. Example:

```
aVar := oObject:callMethod(...)
aVar ->  { VT_..., ... }
```

This means we must reference the second element to obtain the data:

```
xVar := aVar[2]
```

or directly:

```
xVar := oObject:callMethod(...)[2]
```

Here is a real example from our Excel exercise:

```
lDidPrint := oChart:callMethod("PrintOut")[2]
```

However, there is an exception to this rule. If the return value of *:getProperty()* or *:callMethod()* is an OCX object, the array normally returned, such as { VT_DISPATCH, nHandle }, is internally converted to an Xbase^{++} OCX object. Having OCX return objects instead of arrays allows message chaining (the in-line syntax where we "chain" multiple calls to methods in a single line of code):

```
oXbp := XbpSle():new():create()
```

Here is an example using XPPOCX, which results in a ready to use workbook object:

```
oBook := oExcel:GetProperty("Workbooks"):callMethod("Open",cInputTable)
```

Back to the source code. For the first steps with Excel and its usage as an OLE Automation Server, I recommend this code to switch the *:visible* flag ON. This way you can see the result of each step during debugging .

```
#ifdef DEBUG
   oExcel:SetProperty( "Visible", .T. )
#endif
```

Excel is now prepared to process commands.

I will now take each line of the VBA macro code from above and show how to write it using Xbase^{++} XPPOCX syntax. Beginning with the first line:

```
08:   ChDir "H:\Alaska\OCX\Tutorial2"
```

There is no need to translate line #8 to Xbase^{++} code as we fully qualify the path of the table.

```
09:   Workbooks.Open Filename:="H:\Alaska\OCX\Tutorial\SALES.DBF"
```

To help understand this line, here are some notes on VBA syntax:

1) *Workbooks* is a property (in Xbase^{++} terms it is an iVar). We obtain references to properties with the *:getProperty()* method. In this case, the *Workbooks* property contains an object which has an *Open* method.

2) The "." is the same as Xbase^{++}'s ":" operator

3) *Open* is a method and it gets passed a parameter. Methods must be called using *:callMethod()*.

4) It's not in line #9 but the underscore "_" is interpreted as line-continuation character, equivalent to Xbase^{++}'s ";".

In XPPOCX, calls to the OCX object are addressed to the application object that we created at the beginning of the Main() procedure (*oExcel* in our example). So our version of line #9 of the VBA code looks like this:

```
oBook := oExcel:GetProperty("Workbooks"):callMethod("Open",cInputTable)
```

Note: I have to admit that this call looks a bit strange. I personally would prefer a call such as

```
oBook := oExcel:Workbooks:Open( cInputTable )
```

and in fact this does work as well. But it is not recommended yet because XPPOCX is still under development and may change. So, use it at your own risk.

```
10:   Charts.Add
```

Charts is an iVar containing an object. *Add* is a method, and the return value is a *Chart* object:

```
oChart := oExcel:GetProperty( "Charts" ):callMethod( "Add" )
```

```
11:   ActiveChart.ChartType = xl3DPieExploded
```

At this point I am changing the code and workflow a bit. Because I saved the *ActiveChart* object to *oChart*, I do not need to call *oExcel:ActiveChart*. Just a reference to *oChart* will do.

The purpose here is to assign a value to the *ChartType* ivar, but what is the value of *xl3DPieExploded*? I searched the Excel HTML help file and found a list of values under the topic XlChartType enumeration. *Enumeration* is equivalent to our #define directive, so I added a few from the list for our example code:

```
#DEFINE xlPieOfPie            68
#DEFINE xlPieExploded         69
#DEFINE xl3DPieExploded       70
```

So our code is:

```
oChart:SetProperty( "ChartType", xl3DPieExploded )
```

Note: All Excel related #defines which appear in this article were found in the Excel HTML help file (which we generated from the type library).

```
12:   ActiveChart.SetSourceData Source:=Sheets("SALES").Range("A2:B5"),
13:      PlotBy:=xlColumns
```

This command is more complex and I had to split it into several lines.

Note: In VBA, some parameters are not enclosed in parentheses. They are named and comma-separated, and have values assigned to the names, all in the same executable line as the method call. So in the above line, the method *.setSourceData* has two parameters, *Source* and *PlotBy*.

The tricky part here is the parameter *Source:=Sheets("SALES").Range("A2:B5")*, which has to be prepared more thoroughly. The final value of *Source* is a range object created from a worksheet object created from the Excel object. Here is how we arrive at that:

1) *Sheets("SALES")* is quite easy, a method which returns a worksheet object named "SALES":

```
oSheet := oExcel:callMethod( "Sheets", "SALES" )
```

2) Defining the range looks also quite simple:

```
cRange := "A2:B" + ALLTRIM( STR( nRecords + 1 ) )
      (nRecords is from our DBF table)
```

3) Calling the *Range* method of the *Sheet* object is again simple, and results in a range object:

```
oRange := oSheet:callMethod( "Range", cRange )
```

Now we face a new issue: how to perform the *SetSourceData* method call? *xlColumns* is a defined constant. But how do we pass the *Range* object as a parameter?

Earlier in this article we discussed the situation where a return value of a method is an array containing a handle to an object ({VT_DISPATCH,nHandle}), and how XPPOCX internally translates this into an Xbase^{++} OCX object. Now we must do the reverse, and translate our *oRange* into a form which *SetSourceData* can accept. This is done with the same array of two elements, as discussed earlier. The syntax for this array is

```
{ VT_UNKNOWN, oRange:GetPunk() }
```

The constant VT_UNKNOWN (found in XPPOCX.CH) means "pointer unknown". *:getPunk()* is a method in all ActiveX objects which returns a handle to the object (this is similar to our *:getHWND()* method in Xbase^{++}).

So the final line looks like this:

```
#DEFINE xlColumns   2   // from the Excel help file

oChart:callMethod( "SetSourceData", ;
                { VT_UNKNOWN, oRange:GetPunk() }, xlColumns )
```

Here is the complete translation:

```
12:   ActiveChart.SetSourceData Source:=Sheets("SALES").Range("A2:B5"),
13:      PlotBy:=xlColumns
```

XPPOCX:

```
#DEFINE xlColumns   2

oSheet := oExcel:callMethod( "Sheets", "SALES" )

cRange := "A2:B" + ALLTRIM( STR( nRecords + 1 ) )

oRange := oSheet:callMethod( "Range", cRange )

oChart:callMethod( "SetSourceData", ;
                   { VT_UNKNOWN, oRange:GetPunk() }, xlColumns )
```

The next VBA instruction creates a chart in a new work sheet:

```
14:   ActiveChart.Location Where:=xlLocationAsNewSheet,
15:      Name:="PrintGraph"
```

This is equivalent to a method call with two parameters:

```
#DEFINE xlLocationAsNewSheet  1   // from the Excel help file

oChart:callMethod("Location",xlLocationAsNewSheet,"PrintGraph")
```

The following lines of VBA script code show a syntax that is not available in Xbase^{++}. It's a shortcut so you don't have to type the common part of a nested object repeatedly:

```
16:   With ActiveChart
17:      .HasTitle = True
18:      .ChartTitle.Characters.Text = "Xbase++ ActiveX Sample"
19:   End With
```

Is the same as:

```
ActiveChart.HasTitle = True
ActiveChart.ChartTitle.Characters.Text = "Xbase++ ActiveX Sample"
```

This VBA construct allows multiple ivar assignments. Notice line #18 which looks like several nested properties:

```
.ChartTitle.Characters.Text
```

ChartTitle is an ivar containing an object. *Characters* is an ivar of *ChartTitle* and also contains an object, which in turn contains the *Text* ivar. Because these objects will be returned as Xbase^{++} objects by *:GetProperty()*, our XPPOCX version can use the "message chaining" technique we discussed earlier:

```
oChart:SetProperty( "HasTitle", .T. )

oChart:GetProperty( "ChartTitle" ):   ;
       GetProperty( "Characters" ):   ;
       SetProperty( "Text", "Xbase++ ActiveX Sample" )
```

Looking into the Excel help file (which we generated from the Excel type library), I found out that it can be shortened to

```
oChart:GetProperty("ChartTitle"):SetProperty( "Text",;
                  "Xbase++ ActiveX Sample" )
```

The next line instructs Excel to create a legend for the chart:

```
20:   ActiveChart.HasLegend = True
```

The XPPOCX version of line #20 reads:

```
oChart:SetProperty( "HasLegend", .T. )
```

The position of the legend required a mouse click within Excel, and this is recorded in the VBA code:

```
21:   ActiveChart.Legend.Select
22:   Selection.Position = xlCorner
```

These two lines position the legend of the *Chart* object. The first line selects the legend, which is an object, as if it were clicked. The second line assigns a value to its *Position property*. Both lines can be combined as:

```
#DEFINE xlCorner   2   // from the Excel help file

oChart:GetProperty("Legend"):SetProperty("Position",xlCorner)
```

Finally, I wanted the chart to be printed along with data labels, and this is how it looks in VBA code:

```
23:   ActiveChart.ApplyDataLabels Type:=xlDataLabelsShowValue,
24:      LegendKey:=False,HasLeaderLines:=True
```

Line #23 and #24 form a single method call with three parameters. It is translated to Xbase^{++} like this:

```
#DEFINE xlDataLabelsShowValue  2   // from the Excel help file

   oChart:callMethod( "ApplyDataLabels", xlDataLabelsShowValue, .F., .T. )
```

The creation of the chart is now complete and we are getting to the definition of printer options and print the chart:

```
25:   ActiveWindow.SelectedSheets.PrintOut Copies:=1, Collate:=True
```

In Xbase++, this line of code sends the chart to the printer:

```
oChart:callMethod( "PrintOut" , 1, .T. )
```

At this point we should find a page waiting for us on the printer and we have to do the clean-up:

```
26:   ActiveWorkbook.Close
```

This is a single line in VBA code, but it is not sufficient for Xbase++. Since we had to initialize and prepare Excel, we also need to de-initialize and release it. The Excel macro recorder cannot record the shut down of Excel. So, I searched the Excel help files created from the type library for appropriate methods that would be suitable for a graceful shutdown of Excel. This is what I came up with:

```
IF FILE( cExcelFile )
   FERASE( cExcelFile )
ENDIF

oBook:callMethod( "SaveAs", cExcelFile )
oBook:callMethod( "Close", "SaveChanges", .F. )

oExcel:GetProperty( "Workbooks" ):callMethod( "Close" )
oExcel:callMethod( "Quit" )
oExcel:Destroy()
```

This Xbase++ code creates an XLS file (remember, I opened a DBF file with Excel!), closes the work sheet, shuts down Excel and releases the ActiveX component.

Runtime Errors

Runtime errors can occur for various reasons, such as calling an unknown method, omitting parameters or passing invalid parameters. For example, this line results in an Xbase++ runtime error:

```
oBook := oExcel:Workbooks:Open()
```

Xbase++ displays a "parameter not optional" error, because no file name is passed to the *:open()* method. Note, however, that it is not Xbase++ which produces the error message. The XbpOcx() class retrieves error messages from the ActiveX component. Xbase++ doesn't even know what kind of errors may occur with ActiveX components. So keep in mind that all error messages which pop up in the context of ActiveX usage are implemented within the ActiveX component. If you get an unclear message, please contact the producer of that component.

Summary

The ActiveX library from Alaska Software opens up a huge source of existing applications whose functionality can be included into Xbase^{++} applications. I am not an expert in ActiveX yet, but I see its potential after having completed a simple project using a common MS Office application (Excel) as ActiveX component. The potential is tremendous! Of course, using ActiveX components requires to learn the features and interfaces of an ActiveX component. But that's no rocket science! If you know how to approach ActiveX, many tasks can be solved easily without re-inventing the wheel.

My task was "Print a pie chart from data stored in a DBF file using MS Excel's features". I hope to have demonstrated an easy way for you, how to discover ActiveX technology, how to learn about ActiveX interfaces and how to incorporate ActiveX components into your own Xbase^{++} applications. I bet that most of your customers use the MS Office software, so why don't you take advantage of that software? All you need to do is:

* Get an OLE Viewer (e.g. OVI386.EXE)
* Get a program that translates a Type Library to Online Help (e.g. comassistantsetup.exe)
* Take advantage of a Macro Recorder
* Translate VBA script code to Xbase^{++} code

Well, that's how I approached the ActiveX technology and how I learned what becomes possible with Alaska Software's new ActiveX library. I have written this article to share my experiences with you. Please keep in mind, though, that the current ActiveX library is a pre-release version and is still under development. That's why I have used a programming syntax in this article that is less comfortable than the one that is intended to work in future.

```
// guaranteed today and in future
   oExcel:SetProperty( "Visible", .T. )

   // not guaranteed today, but works in future
   oExcel:Visible := .T.
```

Disregarding the programming syntax, I believe that the ActiveX library is a new milestone in Alaska Software's technology. You can incorporate software into your applications that is neither developed by your own nor by Alaska Software. For example, if your customer owns an MS Office package, you can use "Bill's" software in your applications.

If you want to read more about OCX, ActiveX and OLE automation, here is a link to an article which gives you detailed information:

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-
us/dno2kta/html/offaut.asp
```

Best regards,

```
Frank++
Technical Support
Alaska Software
```

Applies to	Xbase^{++} version 1.8, "Goodies"
Keywords	Xbase^{++}, ActiveX, Goodies
Related Links	http://www.alaska-software.com
Copyright	
Last time reviewed	2002-09-15