Broker ActiveX Control with Visual Basic

Visual Basic is used here as an example of a development environment in which applications using Broker ActiveX Control can work. Broker ActiveX Control can be used by any programming language or programming environment that can act as a container for ActiveX controls.

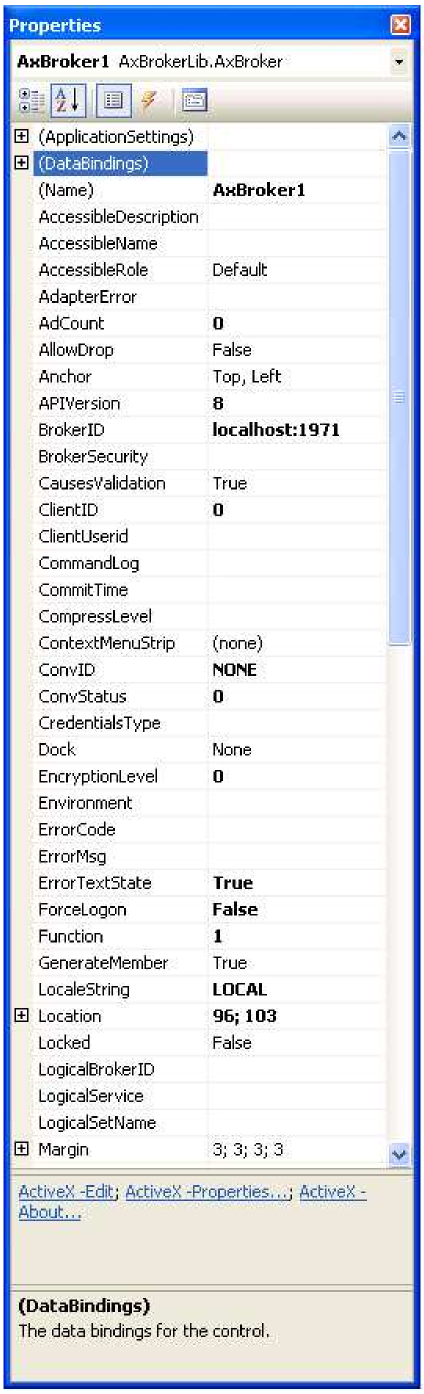
Note:
If you edit a Visual Basic application that uses Broker ActiveX Control and save these changes with the new version of Broker ActiveX Control, you will not be able to use this application with Broker ActiveX Control version 1.2.1.

This chapter covers the following topics:

Step 1: Instantiate EntireX Broker ActiveX Control

To use Broker ActiveX Control as a control

1. From the **Project, Components, Controls** menu choose **EntireX Broker ActiveX Control**.
2. Drop it into your dialog.

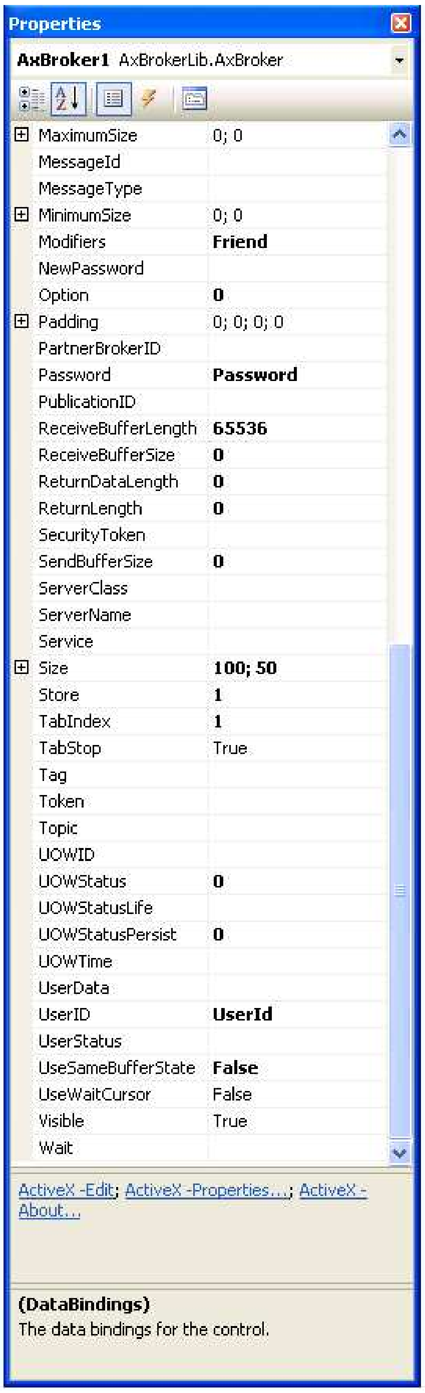


AxBroker1 AxBrokerLib.AxBroker

⊞ (ApplicationSettings)	
⊞ (DataBindings)	
(Name)	**AxBroker1**
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AdapterError	
AdCount	**0**
AllowDrop	False
Anchor	Top, Left
APIVersion	**8**
BrokerID	**localhost:1971**
BrokerSecurity	
CausesValidation	True
ClientID	**0**
ClientUserid	
CommandLog	
CommitTime	
CompressLevel	
ContextMenuStrip	(none)
ConvID	**NONE**
ConvStatus	**0**
CredentialsType	
Dock	None
EncryptionLevel	**0**
Environment	
ErrorCode	
ErrorMsg	
ErrorTextState	**True**
ForceLogon	**False**
Function	**1**
GenerateMember	True
LocaleString	**LOCAL**
⊞ Location	**96; 103**
Locked	False
LogicalBrokerID	
LogicalService	
LogicalSetName	
⊞ Margin	3; 3; 3; 3

ActiveX -Edit; ActiveX -Properties...; ActiveX - About...

(DataBindings)
The data bindings for the control.



AxBroker1 AxBrokerLib.AxBroker

⊞ MaximumSize	0; 0
MessageId	
MessageType	
⊞ MinimumSize	0; 0
Modifiers	**Friend**
NewPassword	
Option	**0**
⊞ Padding	0; 0; 0; 0
PartnerBrokerID	
Password	**Password**
PublicationID	
ReceiveBufferLength	**65536**
ReceiveBufferSize	**0**
ReturnDataLength	**0**
ReturnLength	**0**
SecurityToken	
SendBufferSize	**0**
ServerClass	
ServerName	
Service	
⊞ Size	**100; 50**
Store	**1**
TabIndex	**1**
TabStop	True
Tag	
Token	
Topic	
UOWID	
UOWStatus	**0**
UOWStatusLife	
UOWStatusPersist	**0**
UOWTime	
UserData	
UserID	**UserId**
UserStatus	
UseSameBufferState	**False**
UseWaitCursor	False
Visible	True
Wait	

ActiveX -Edit; ActiveX -Properties...; ActiveX - About...

(DataBindings)
The data bindings for the control.

In this example, Name is set to "BOX" in the **Properties** dialog:

Using Broker ActiveX Control as an Automation Server:

If you want to

- see the interface description of Broker ActiveX Control in the object browser or
- use the early bind feature,

from the **Project > References** menu choose **Browse** and then select Broker ActiveX Control in *<drive>:\SoftwareAG\EntireX\bin\ebx.dll*.

To use Broker ActiveX Control as an automation server, you can define the following in your code:

```
Dim BOX as Object
```

or

```
Dim BOX as Broker
Set BOX=CreateObject("EntireX.Broker.ACI")
```

If you use Broker ActiveX Control as an automation server, you will not be able to:

- call the methods `DefineTOMethods` and `AboutBox`
- use the property pages.

Step 2: Instantiate the Transaction Object

If a Transaction Object Repository (TOR) file is used, it is not necessary to set the other properties. If you want to use a transaction object, instantiate the transaction object with the command:

```
Dim TransObject As Object
Set TransObject = BOX.CreateTransObject("c:\\path\\to\\trans\\object\\object.tor")
```

BOX is the name set previously.

See the list of methods available for supporting transaction objects.

Step 3: Call Methods

Once a transaction object has been instantiated, the methods defined in that transaction object can be called. If the transaction object method being called has one or more return values, transaction object methods *always* return these values wrapped in a return object.

```
Dim ReturnObject As Object
Set ReturnObject = TransObject.MyMethod("Param1", 50, "Param3")
```

A return object is always used, as TO methods usually return multiple scalar data items, or arrays, structures or records. These in fact define the possible return values in a return object. They will be either scalars:

- 2-byte INT
- 4-byte INT
- etc., basically all scalar types handled through the automation VARIANT structure

or objects:

- structure objects

- collection objects
- arrays
- records

Alias custom types are mapped internally to the data type they alias, either scalars or objects.

Step 4: Access the Returned Data

You then access the returned data by interpreting the return object. The code required depends on whether you are accessing scalars, structures, or arrays and records.

Note:
Care must be taken to avoid recursive complex type definitions. For example, a structure should not be defined that contains an instance of itself, or less directly, an array of structures should not be defined that contains an instance of the same array type. These and other permutations of recursive definitions cannot be resolved, and thus cannot be used.

Scalars

Scalars can be accessed through the return object with code like this:

```
Dim Str As String
Dim Int As Integer
Str = ReturnObject.MyString
Int = ReturnObject.MyInt
```

Structures

Structures can be accessed from the return object like this:

```
Dim Struct As Object
Dim Str As String
Set Struct = ReturnObject.MyStruct
Str = Struct.MyString
```

Arrays and Records Exposed as Collections

Arrays and records are exposed by Broker ActiveX Control as automation collections when the method `CreateTransObject` is used. As collections, they support the `Count` property, as well as the `Item` property that acts as the default value when subscripting is performed without the `Item` name. Thus, an array in the return object can be accessed like this:

```
Dim Array_Value As Object
Dim I As Integer
Dim MyInt As Integer
Set Array_Value = ReturnObject.MyArray
For I = 0 To Array_Value.Count - 1
  MyInt = Array_Value(I)
Next I
```

The elements of a record can be accessed with the following method:

```
Dim Array_Value,Struct As Object
Dim I As Integer
Set Array_Value = ReturnObject.MyArray
For I = 0 To Array_Value.Count - 1
  Set Struct = Array_Value(I)
  Str = Struct.Str
Next
```

or also:

```
Dim Array_Value,Struct As Object
Dim I As Integer
Set Array_Value = ReturnObject.MyArray
For Each Struct in Array_Value
  Str = Struct.str
Next
```

Arrays and Records Exposed as Safe Arrays

Arrays and Records are exposed as safe arrays when the method `CreateTransObjectSA(`*`torfilename`*`)` is used. Instead of the `Count` property, the `LBound` and `UBound` functions are supported.

An array in the return object can be accessed like this:

```
Dim Array_Value as Variant
Dim I as Integer
Dim Str as String

Array_Value = ReturnObject.MyArray
For I = LBound(Array_Value) To UBound(Array_Value)
  Str = Array_Value[I]
Next
```

The elements on a record can be accessed with the following method:

```
Dim Array_Value as Variant
Dim Struct as Variant
Dim I as Integer
Dim Str as String

Array_Value = ReturnObject.MyArray
For I = LBound(Array_Value) To UBound(Array_Value)
  Set Struct = Array_Value[I]
  Str = Struct.Str
Next
```

Another possible `For` statement:

```
For Each Struct in Array_Value
  Str = Struct.Str
Next
```

There are no limitations to the number of complex types or their relationship to each object in a transaction object. Arrays can exist within structures, and conversely, structures and arrays can exist within records, etc. Thus, multidimensional arrays can easily be simulated if the given Broker service that the method maps to provides data in such a format.

Step 5: Cleanup Resources

When objects in your automation code are no longer used, be sure to call:

```
Set ObjectName = Nothing
```

This decrements the reference count of the object, thus allowing cleanup of object resources. While the above information pertains specifically to Visual Basic, the concepts are also relevant to other automation controllers, such as Delphi and FoxPro.

Step 6: Error Handling in Transaction Object Methods

TO methods do not return an error flag; they raise a standard ActiveX exception instead. In Visual Basic, this exception can be caught with an ’On error’ clause. The most likely reason for the failure of a TO method is that the Broker call that was issued returned an error. In Visual Basic, use the standard Err object to retrieve the error number and message (Err.Number and Err.Description).

If the error is a Broker error, Err.Description shows a generic error message "Automation Error". For a detailed error description use the `ErrorCode` and `ErrorMsg` properties.

Examples: Writing an ACI Client and Server with Broker ActiveX Control

- Writing an ACI Client with Broker ActiveX Control
- Writing an ACI Server with Broker ActiveX Control

Writing an ACI Client with Broker ActiveX Control

```
On Error Resume Next
Dim ebx As Object
Dim senddata As String
Dim loopcount As Integer

 loopcount = 0
’ simple data to send
senddata = "Hello"

 Set ebx = CreateObject("EntireX.Broker.ACI")
ebx.BrokerID = "localhost"
ebx.ServerClass = "ACLASS"
ebx.ServerName = "ASERVER"
ebx.Service = "ASERVICE"
ebx.UserId = "EBX-USER"

ebx.function = 9 ’ Logon
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMessage
Exit Sub
End If

Do
ebx.function = 1 ’ Send
ebx.ConvID = "NONE"
’ SetSendData data, length of data
ebx.SetSendData senddata, Len(senddata)
ebx.wait = "10s" ’ wait 10 seconds for a response from server
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMsg
Else
MsgBox "Received " + Str(ebx.ReturnDataLength) + " bytes (" + ebx.GetReceiveData + ")"
End If
loopcount = loopcount + 1
If loopcount = 2 Then
senddata = " shutdown"
End If

Loop Until loopcount > 2

ebx.function = 10 ’ Logoff
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMessage
End If
```

Writing an ACI Server with Broker ActiveX Control

```
On Error Resume Next

Dim ebx As Object
Dim senddata As String
Dim receivedata As String

’ simple data to send
senddata = "Hello"

Set ebx = CreateObject("EntireX.Broker.ACI")
ebx.BrokerID = "localhost"
ebx.ServerClass = "ACLASS"
ebx.ServerName = "ASERVER"
ebx.Service = "ASERVICE"
ebx.UserId = "EBX-USER"

ebx.function = 9 ’ Logon
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMessage
Exit Sub
End If

ebx.function = 6 ’ Register
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMessage
End If

Do
ebx.function = 2 ’ Receive
ebx.wait = "yes" ’ wait until data is received
ebx.ConvID = "NEW"
ebx.SetReceiveBufferLength = 1024 ’ we are now able to receive messages up to 1024 bytes
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMsg
```

```
Else

’ save received data
receivedata = ebx.GetReceiveData
’ send response
ebx.function = 1 ’ Send
’ SetSendData data, length of data
ebx.SetSendData senddata, Len(senddata)
ebx.wait = "no" ’ don’t wait for a response
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMsg
Else
MsgBox "Received data: " + receivedata
End If
End If
’ loop until the received data has the string "shutdown" from the position 20
receivedata = Mid(receivedata, 20, 8)
Loop Until receivedata = "shutdown"

ebx.function = 7 ’ DeRegister
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMessage
End If

ebx.function = 10 ’ Logoff
ebx.InvokeBrokerFunction
If ebx.ErrorCode <> 0 Then
MsgBox ebx.ErrorMessage
End If
```