

webMethods EntireX

EntireX Broker ActiveX Control

Version 9.6

April 2014

This document applies to webMethods EntireX Version 9.6.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 1997-2014 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors..

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA, Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at <http://documentation.softwareag.com/legal/>.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at <http://documentation.softwareag.com/legal/> and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices and license terms, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". This document is part of the product documentation, located at <http://documentation.softwareag.com/legal/> and/or in the root installation directory of the licensed product(s).

Document ID: EXX-EEXXACI-96-20140628ACTX

Table of Contents

Preface	v
1 Broker ActiveX Control Introduction	1
Broker ACI	2
Transaction Objects	2
2 Writing Applications - Broker ActiveX Control	3
Calling a Broker Function	4
Viewing the Type Library	6
Adding the Broker ActiveX Control Component to Visual Studio	7
Using Internationalization with Broker ActiveX Control	9
Using the Property Pages	10
3 Broker ActiveX Control with Visual Basic	13
Step 1: Instantiate EntireX Broker ActiveX Control	14
Step 2: Instantiate the Transaction Object	16
Step 3: Call Methods	16
Step 4: Access the Returned Data	17
Step 5: Cleanup Resources	20
Step 6: Error Handling in Transaction Object Methods	20
Examples: Writing an ACI Client and Server with Broker ActiveX Control	20
4 Using Broker ActiveX Control with Active Server Pages	25
Prerequisites	26
Designing a Web Page with ASP and Broker ActiveX Control	26
Using Broker ActiveX Control in Multiple Pages	28
5 Using Broker ActiveX Control with .NET	29
Using Broker ActiveX Control with Visual Studio .NET	30
A Small Visual Basic .NET Example	30
6 Transaction Objects in Broker ActiveX Control	31
Advantages of Transaction Objects	32
Calling the Transaction Object Editor	32
Managing TOR Files	34
Defining Methods	37
Specifying Connection Information	43
Defining Custom Data Types	46
TOR Files in IDL Format	49
TOR Files in XML Format	51
Storing TOR Files in a Tamino Database	55
7 Calling Broker ActiveX Control Remotely	59
Setting up the Server Environment	60
Setting up the Client Environment	65
Testing the Connection	68
8 Publish and Subscribe with Broker ActiveX Control	71
Writing Subscriber Applications	72
Writing Publisher Applications	77
9 Reference - Broker ActiveX Control	81

Methods of Broker ActiveX Control	82
Properties of Broker ActiveX Control	83

Preface

Broker ActiveX Control allows GUI application developers to use an ActiveX-based interface to access EntireX Broker. It can be used within ActiveX containers, such as Visual Basic, PowerBuilder, Delphi, Microsoft Excel, Microsoft Word.

Broker ActiveX Control Introduction

Broker ActiveX Control provides a programmatic interface to COM-enabled programming environments. It has two types of operation: the Broker ACI and transaction objects. Broker ActiveX Control enables you to create EntireX ACI clients and EntireX ACI servers.

Writing Applications - Broker ActiveX Control

Topics include calling a Broker function; viewing the type library; using internationalization; using property pages.

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.

Using Broker ActiveX Control with Active Server Pages

Microsoft's Active Server Page (ASP) is an HTML page that includes one or more scripts and reusable ActiveX server components to create dynamic Web pages. The scripts and ActiveX components are processed on a Microsoft Web server before the page is sent to the user.

Using Broker ActiveX Control with .NET

How to use Broker ActiveX Control with Visual Studio .NET. An example is provided.

Transaction Objects in Broker ActiveX Control

Transaction objects (TOs) in Broker ActiveX Control are selections of logical methods that are stored in a transaction object repository (TOR). These logical methods contain all the connection and interface details necessary to communicate with the Broker.

Calling Broker ActiveX Control Remotely

You can call Broker ActiveX Control remotely if you use it as an automation server. This means you can use the Broker component from a separate process - either on the same machine or on another machine in the network.

Publish and Subscribe with the Broker ActiveX Control

Broker ActiveX Control provides five Broker functions to enable publishing and subscription. Publish and subscribe enables an application to send a message (publication) to multiple receivers (subscribers).

Reference - Broker ActiveX Control

Methods and properties of Broker ActiveX Control.

1 Broker ActiveX Control Introduction

- Broker ACI 2
- Transaction Objects 2

Broker ActiveX Control provides a programmatic interface to COM-enabled programming environments. It has two types of operation: the Broker ACI and transaction objects. Broker ActiveX Control enables the user to create EntireX ACI clients and EntireX ACI servers.

Broker ACI

The Broker ACI provides a simple automation API that is one-to-one compatible with the published EntireX Broker ACI. It provides Broker ActiveX Control properties and corresponding property pages for the control parameters detailed in the Broker ACI fields. This API is conceptually compatible with current Broker programming practices. Further, the Broker ActiveX Control programmer can count on programmatic behavior consistent with programming the Broker API directly, such as non-blocking calls and polling for completion.

Transaction Objects

Broker ActiveX Control generates ActiveX automation server interfaces dynamically at runtime from files in the Transaction Object Repository (TOR).

Broker ActiveX Control transaction objects provide a dictionary subsystem and user interface that will allow the EntireX Broker developer to define a dynamic IDispatch interface. This interface allows received data to be accessed with the traditional automation methodology.

The transaction object definition of a method also includes parsing up the SEND and RECEIVE buffers of a Broker message into parameters and return properties respectively. The transaction objects are loaded at runtime and the ActiveX container can then call the methods of that transaction object to send/receive data.

See [*Transaction Objects in Broker ActiveX Control*](#) for more information.

2 Writing Applications - Broker ActiveX Control

▪ Calling a Broker Function	4
▪ Viewing the Type Library	6
▪ Adding the Broker ActiveX Control Component to Visual Studio	7
▪ Using Internationalization with Broker ActiveX Control	9
▪ Using the Property Pages	10

Calling a Broker Function

Setting the Broker ActiveX Properties

You can set the Broker ActiveX properties either in the program or in the [property pages](#).

Specifying the Send Parameters

Before executing a send function, specify the send parameters with the method `SetSendDataLong(String bsData, Long DataLen)` or `SetSendData(String bsData, Short DataLen)`.

This method sets only the send buffer.

The first parameter specifies the buffer that has to be sent to the server. The second parameter specifies the number of bytes to be transferred.

The following rules apply to the `SetSendData` method:

- The `DataLen` bytes of the string `bsData` are copied to the internal send buffer.
- A byte copy is performed (not a string character copy), which means that the string `bsData` can contain zero bytes.
- The function `BOOL SetSendData(String bsData, Short DataLen)` can be used if the send buffer is smaller than 32 KB.

Calling the Broker Function

- Set the required properties.
- When you use the send function, use the method `SetSendData` to set up the send buffer.
- When you use the receive function, use the property `ReceiveBufferSize` to set up the size of the internal receive buffer.
- Use the static automation method to call the Broker functions:

```
BOOL InvokeBrokerFunction()
```

This method executes the Broker function defined by the current value of the property `Function`. Depending on the function, the required Broker parameters are taken from the current values of the corresponding properties.

If the Broker call is successful:

- The function returns `TRUE`.

- The `ErrorCode` property is set to '00000000' and the `ErrorMsg` property is empty.

If the Broker call is a `Send` or `Receive` function, this call may also update the `ConvID` property.

If the Broker call is a `Receive` function and asterisks were specified for `ServerClass`, `ServerName` and `Service`, the call updates the `ServerClass`, `ServerName` and `Service` properties.

If the Broker call is a `Receive` or `Send` with implicit `Receive` (`Wait > 0`), the number of bytes received is stored in the property `ReturnDataLength` and the returned data can be retrieved with the `GetReceiveData` method.

If the Broker call fails:

- The function returns `FALSE`.
- The `ErrorCode` and `ErrorMsg` properties contain the corresponding error reason.

The error code has two parts:

- error class (first four digits), which provides information for the application on how to react to the returned error, and
- error number (last four digits), which indicates the reason for the error.

The `GetErrorText` method is still available and returns the value of the `ErrorMsg` property.

For more information see *Error Messages and Codes*.

Getting the Contents of the Receive Buffer

If a `Receive` function was executed, the receive buffer can be retrieved with the function

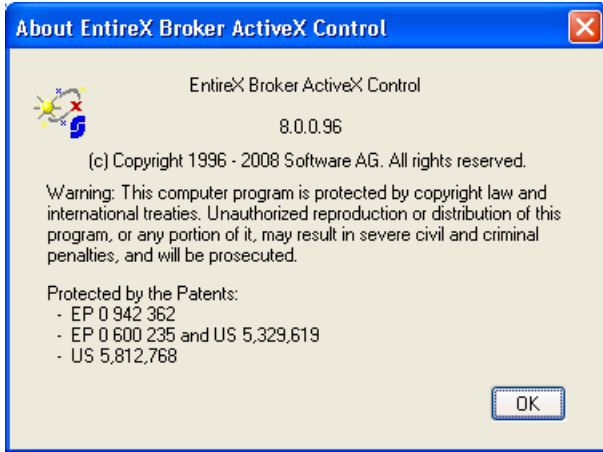
```
STRING GetReceiveData()
```

AboutBox

The `AboutBox` method is used to show the version of Broker ActiveX Control.

A message box will be displayed containing the **About** information.

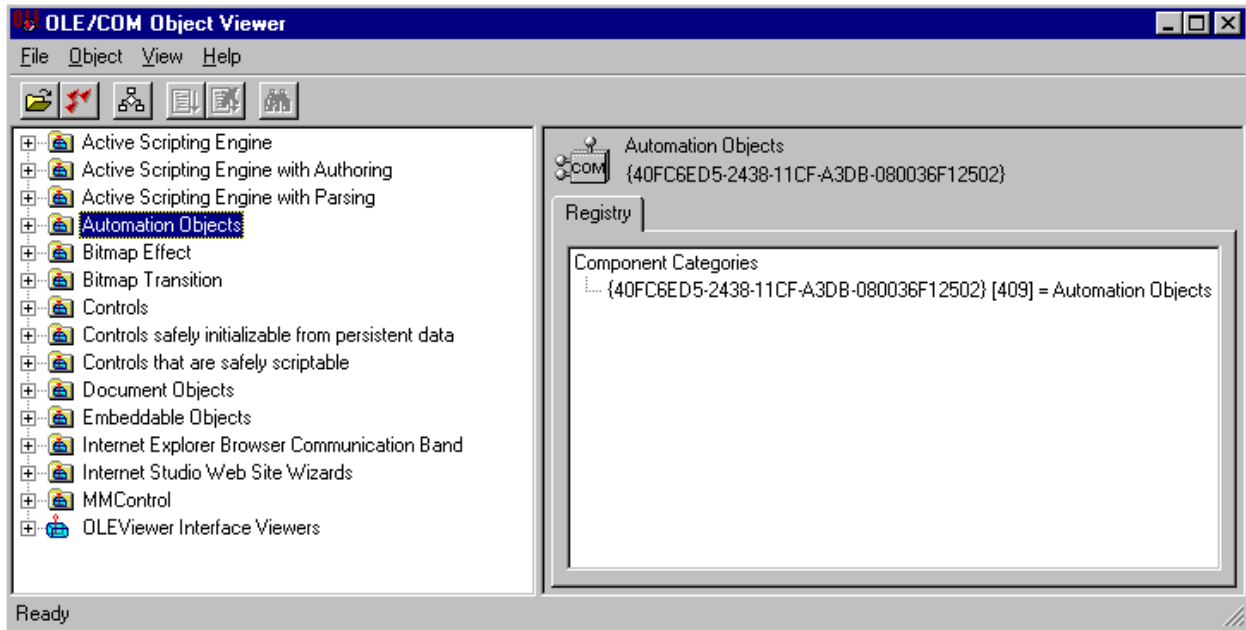
```
AboutBox ( )
```



Viewing the Type Library

► To view the Type Library of Broker ActiveX Control

- Use the OLE/COM Object Viewer (choose **EntireX Broker ActiveX Control** and choose **View Type Information**).

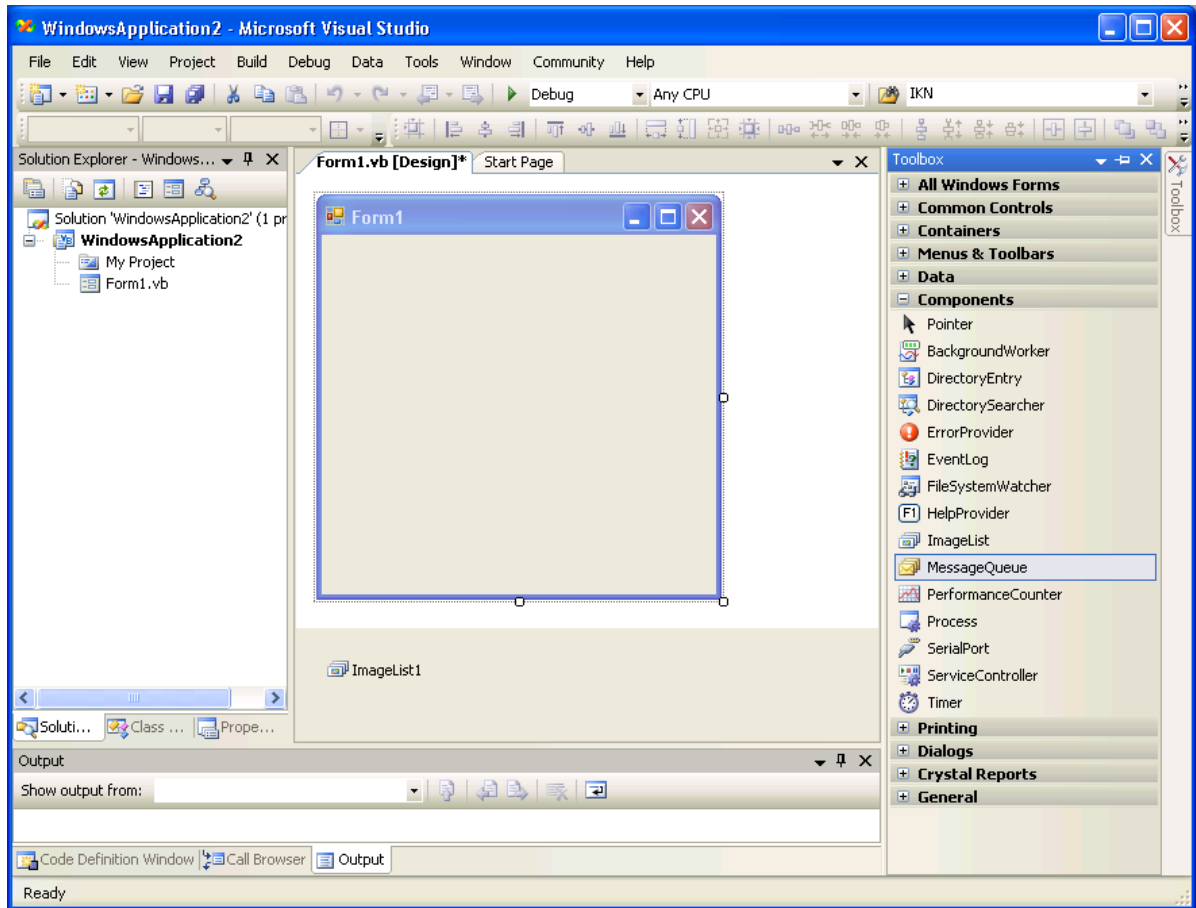


To do this with Visual Basic, see [Using Broker ActiveX Control as an Automation Server](#).

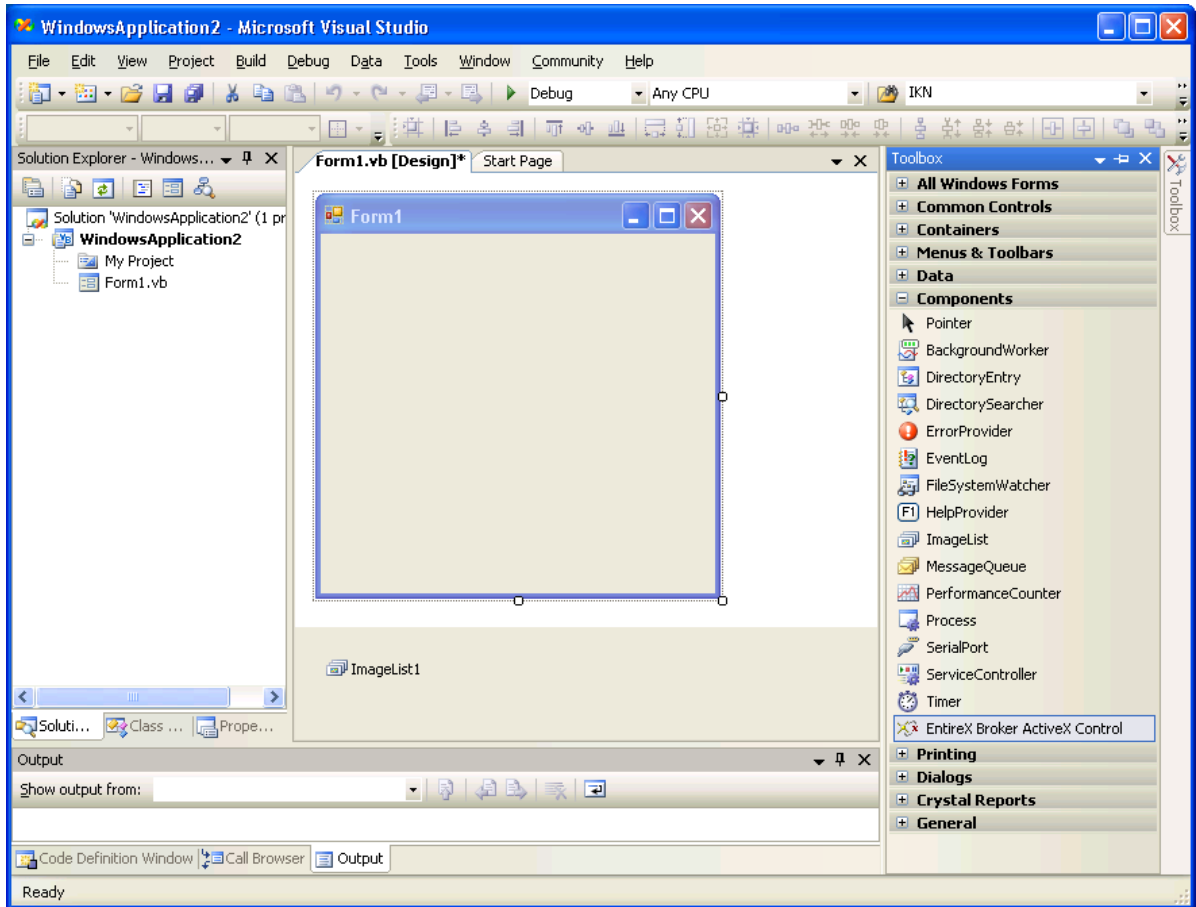
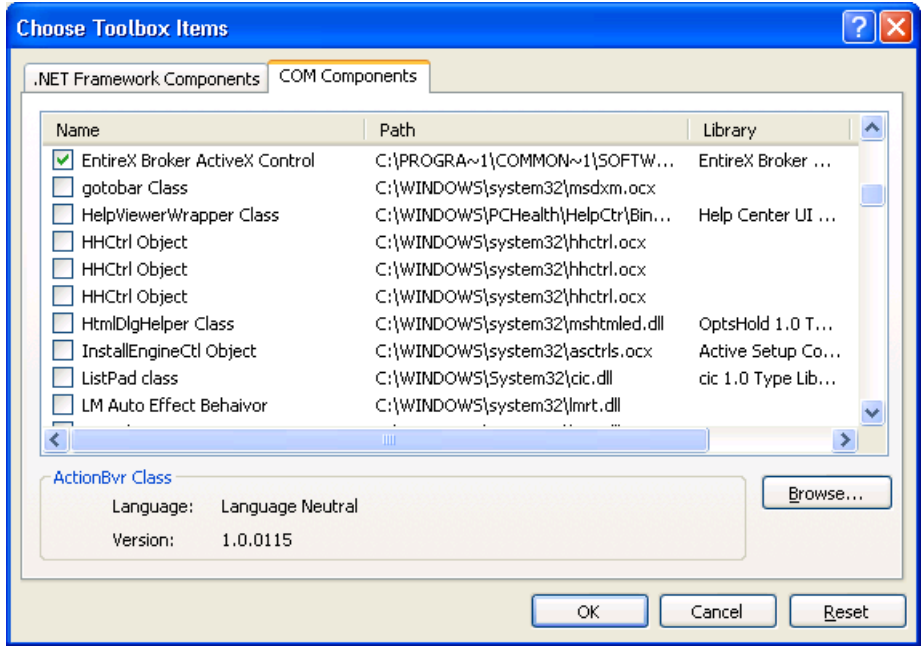
Adding the Broker ActiveX Control Component to Visual Studio

▶ To add the Broker ActiveX Control component to Visual Studio

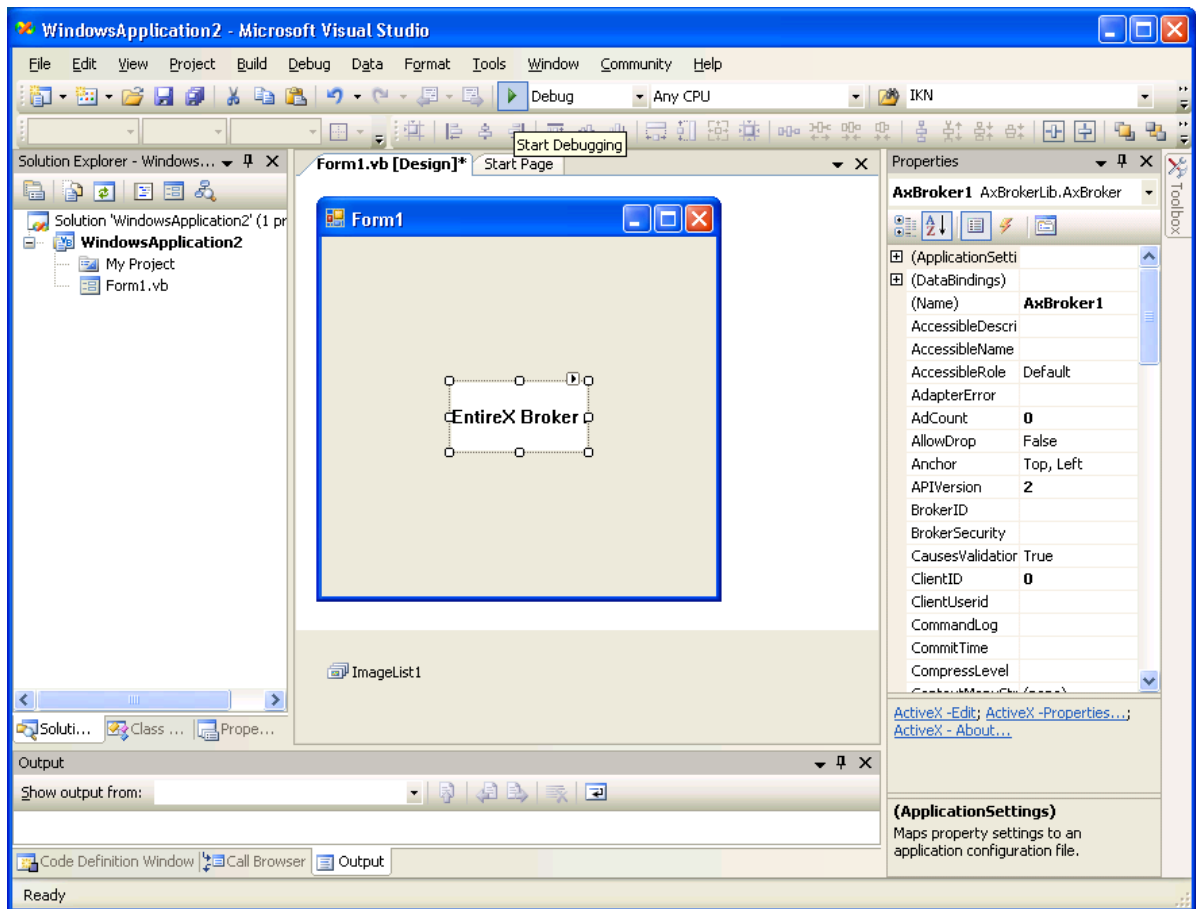
- 1 In Visual Studio, choose **Toolbox > Components**.



- 2 From the context menu, choose **Choose Item**.
- 3 In the **Choose Toolbox Items** dialog under **COM Components**, check "EntireX Broker ActiveX Control".



EntireX Broker ActiveX Control is now known to Visual Studio. It can be copied and pasted into the new form for later use.



Using Internationalization with Broker ActiveX Control

It is assumed that you have read the document *Internationalization with EntireX* and are familiar with the various internationalization approaches described there.

By default, Broker ActiveX Control uses the Windows ANSI codepage to convert the Unicode (UTF-16) representation within BSTRINGS to the multibyte or single-byte encoding sent to or received from the broker. This codepage is also transferred as part of the locale string to tell the broker the encoding of the data.

If you want to adapt the Windows codepage, see the Regional Settings in the Windows Control Panel and your Windows documentation.

With the property `LocaleString` (see [LocaleString](#) in *Reference - Broker ActiveX Control*) you can prevent a locale string from being sent if communicating with broker version 7.1.x and below (blank out the property for this purpose).

Restrictions

- Only the codepage configured for Windows in the Regional Settings can be used. It is not possible to use any codepage other than the codepage configured for Windows in the Regional Settings. Only LOCAL or blank is allowed as a value for the property. See *Using the Abstract Codepage Name LOCAL* under *Locale String Mapping* in the internationalization documentation for more information.
- No TOR file property is available. When you are using the TOR interface, you can set this property as usual in your own application.
- The Windows codepage used by Broker ActiveX Control must also be a codepage supported by the broker, depending on the internationalization approach. See *Locale String Mapping* in the internationalization documentation for information on how the broker derives the codepage from the locale string.

Using the Property Pages

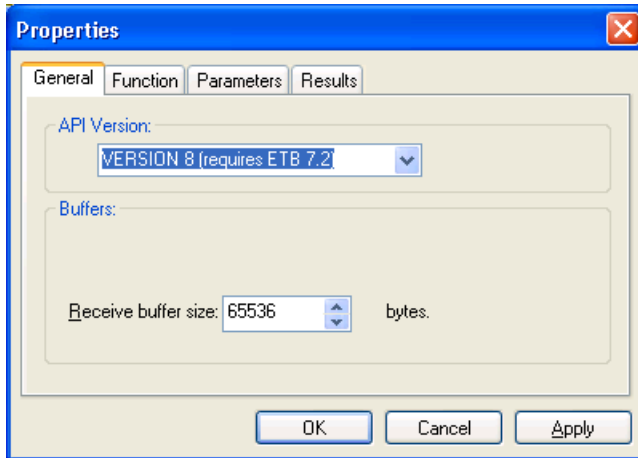
If you do not use Transaction Object Repository (TOR) files, you can also supply the properties using the property sheet of Broker ActiveX Control. (If you use Broker ActiveX Control as an automation server, the property pages are not available.)

The property sheet contains the following:

- [General Page](#)
- [Function Page](#)
- [Parameters Page](#)
- [Results Page](#)

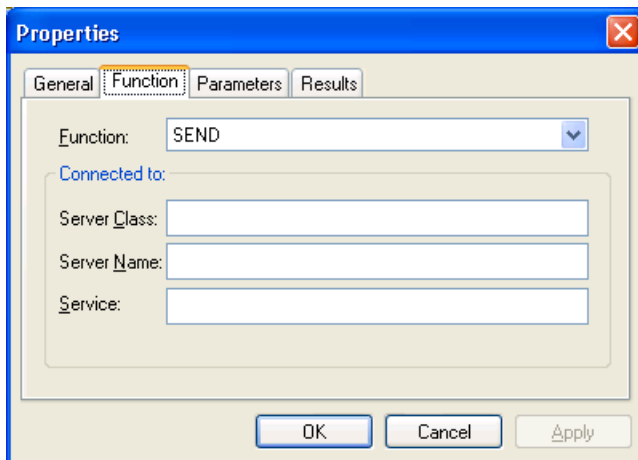
General Page

With this page you can specify the API version and the size of the receive buffer.



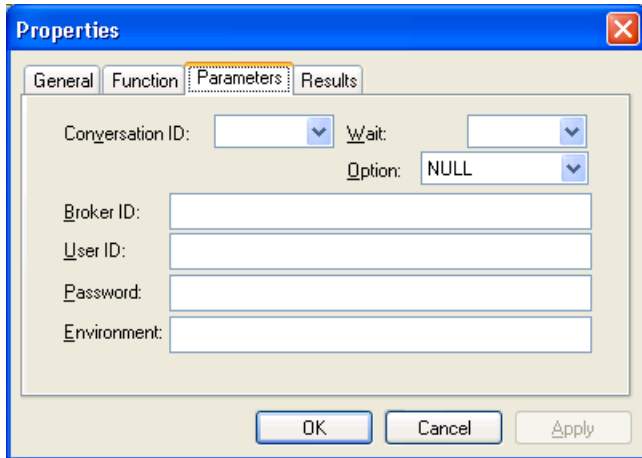
Function Page

With this page you can specify the function to be called and Service, Server Class and Server Name.



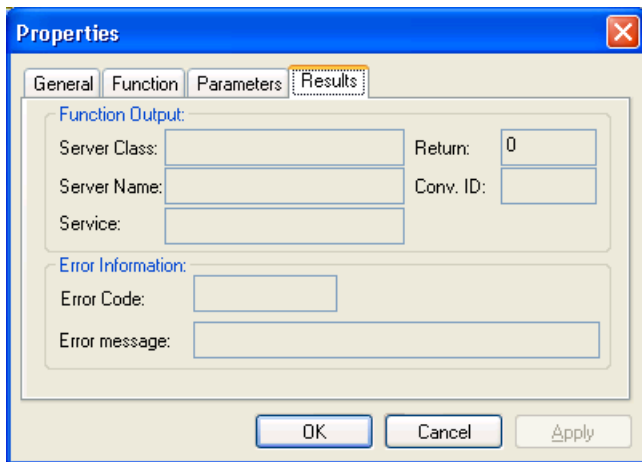
Parameters Page

With this page you can specify the Conversation ID, Broker ID, User ID, Password, Environment, Wait time, and Option.



Results Page

This page displays the results of the Broker function.



3 Broker ActiveX Control with Visual Basic

- Step 1: Instantiate EntireX Broker ActiveX Control 14
- Step 2: Instantiate the Transaction Object 16
- Step 3: Call Methods 16
- Step 4: Access the Returned Data 17
- Step 5: Cleanup Resources 20
- Step 6: Error Handling in Transaction Object Methods 20
- Examples: Writing an ACI Client and Server with Broker ActiveX Control 20

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.

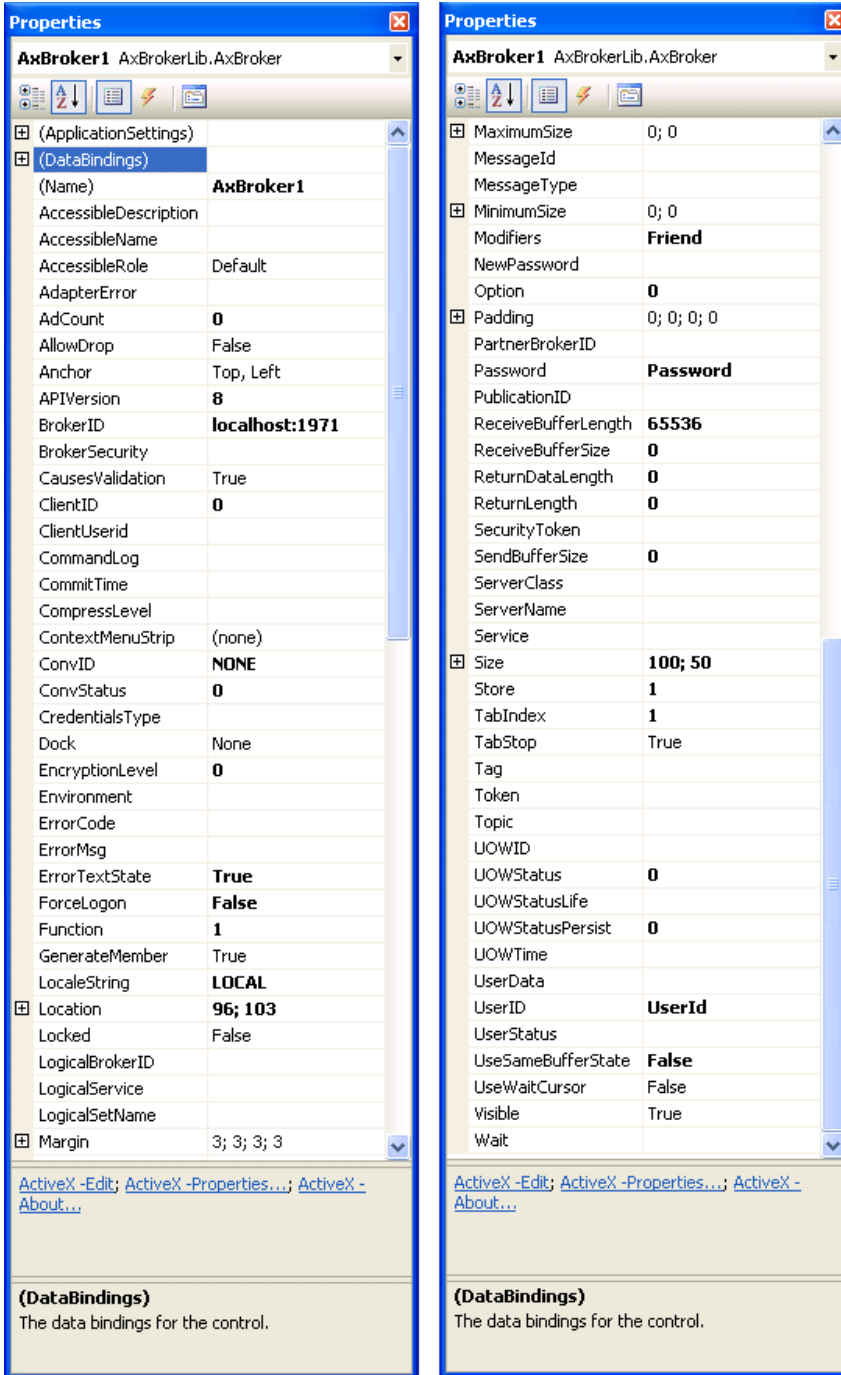


Note: If you edit a Visual Basic application that uses Broker ActiveX Control and save these changes with the new version of Broker ActiveX Control, you will not be able to use this application with Broker ActiveX Control version 1.2.1.

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.



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 = "ACCLASS"
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
```


4 Using Broker ActiveX Control with Active Server Pages

- Prerequisites 26
- Designing a Web Page with ASP and Broker ActiveX Control 26
- Using Broker ActiveX Control in Multiple Pages 28

Microsoft's Active Server Page (ASP) is an HTML page that includes one or more scripts and reusable ActiveX server components to create dynamic Web pages. The scripts and ActiveX components are processed on a Microsoft Web server before the page is sent to the user.

Prerequisites

Installation prerequisites for all EntireX components are described centrally. See *Prerequisites* in the EntireX Release Notes.

To use Broker ActiveX Control with ASP, you must have a running Web server.

Designing a Web Page with ASP and Broker ActiveX Control

Creating an Instance of the ActiveX Control and the Transaction Object

```
<%  
Set EBX = server.Createobject("EntireX.Broker.ACI")  
Set torobj = EBX.CreateTransObject("calc.tor")
```

or

```
Set torobj = EBX.CreateTransObjectSA("calc.tor") (if returnvalue contains array)  
%>
```

Calling a TOR Method

```
Set retobj = torobj.calc(op,op1,op2)
```

Accessing the Data

Scalars

```
<% string = retobj.result %>
```


Structures

```
<% string = retobj.result.str %>
```

Arrays

You can have access to array elements:

```
<%string = retobj.retarr(0) %>
```

or

```
<%  
return = retobj.retarr  
string = return(0)  
%>
```

or

```
<%  
For Each element in retobj.retarr  
  string = element  
Next  
%>
```

Records

You can have access to record elements:

```
<%string = retobj.retrec(0).str %>
```

or

```
<%  
Set return = retobj.retrec(3)  
Response.Write return.str  
%>
```

or

```
<%  
For Each struct in retobj.retrec  
    string = struct.str  
Next  
>
```

or

```
<%  
Array_Value = retobj.retrec  
For I = LBound(Array_Value) To UBound(Array_Value)  
    string = Array_Value(I).str  
Next  
>
```

Using Broker ActiveX Control in Multiple Pages

Objects created by `Server.CreateObject` or `CreateTransObject` have page scope. They will be destroyed automatically when the current ASP page is finished.

To create an object with session or application scope, you can either use the `<OBJECT>` tag and set the `SCOPE` parameter to `SESSION` or `APPLICATION`, or store the object in a session or application variable.

For example, an object stored in a session variable, as shown in the following script, is destroyed when the Session object is destroyed. That is, when the session times out, or the `Abandon` method is called.

```
<% Set Session("torobj") = EBX.CreateTransObject("calc.tor")%>
```

You can destroy the object by setting the variable to "Nothing" or setting the variable to a new value.

```
<% Session("torobj") = Nothing %>
```

5

Using Broker ActiveX Control with .NET

- Using Broker ActiveX Control with Visual Studio .NET 30
- A Small Visual Basic .NET Example 30


Using Broker ActiveX Control with Visual Studio .NET

▶ To use Broker ActiveX Control with Visual Studio .NET

- 1 Add Broker ActiveX Control to the Project references.
- 2 Add a Broker Control variable `BrokerLib.BrokerClass()`.

While you are using Broker ActiveX Control, the properties and methods of the object are listed in the member list.

Using Custom Data Types

 **Important:** To use **custom data types** you have to access the items through a temporary object.

A Small Visual Basic .NET Example

```
' create new ActiveX Control
Dim broker As New BrokerLib.BrokerClass()

Dim TransactionObject As Object
Dim SomeObject As Object
Dim CTOBJECT As Object

' load tor object
TransactionObject = broker.CreateTransObject("Broker.tor")

' call a method from the tor object
SomeObject = TransactionObject.GetData("Person1")

'
reference a temporary object to the Customer Data type

CTOBJECT = SomeObject.CustData

' access to the items of the Customer Data
Console.WriteLine("Name :" & CTOBJECT.Name)
Console.WriteLine("Address :" & CTOBJECT.Address)
```

6 Transaction Objects in Broker ActiveX Control

- Advantages of Transaction Objects 32
- Calling the Transaction Object Editor 32
- Managing TOR Files 34
- Defining Methods 37
- Specifying Connection Information 43
- Defining Custom Data Types 46
- TOR Files in IDL Format 49
- TOR Files in XML Format 51
- Storing TOR Files in a Tamino Database 55

Transaction Object (TOs) in Broker ActiveX Control are selections of logical methods that are stored in a transaction object repository (TOR). These logical methods contain all the connection and interface details necessary to communicate with EntireX Broker.

Advantages of Transaction Objects

The advantages of using transaction objects are:

- Services are defined once, in one place, and distributed as needed. They can then be used by anyone from many different applications to access back-end applications.
- Transaction objects can encapsulate all connection and conversational information from the developer, which simplifies the implementation and administration of distributed applications.
- The SEND-BUFFER of a message is broken down into parameters, and the RECEIVE-BUFFER is mapped to the return object. This means you do not have to worry about offsets, data types, repeating fields (arrays), or structures.

Calling the Transaction Object Editor

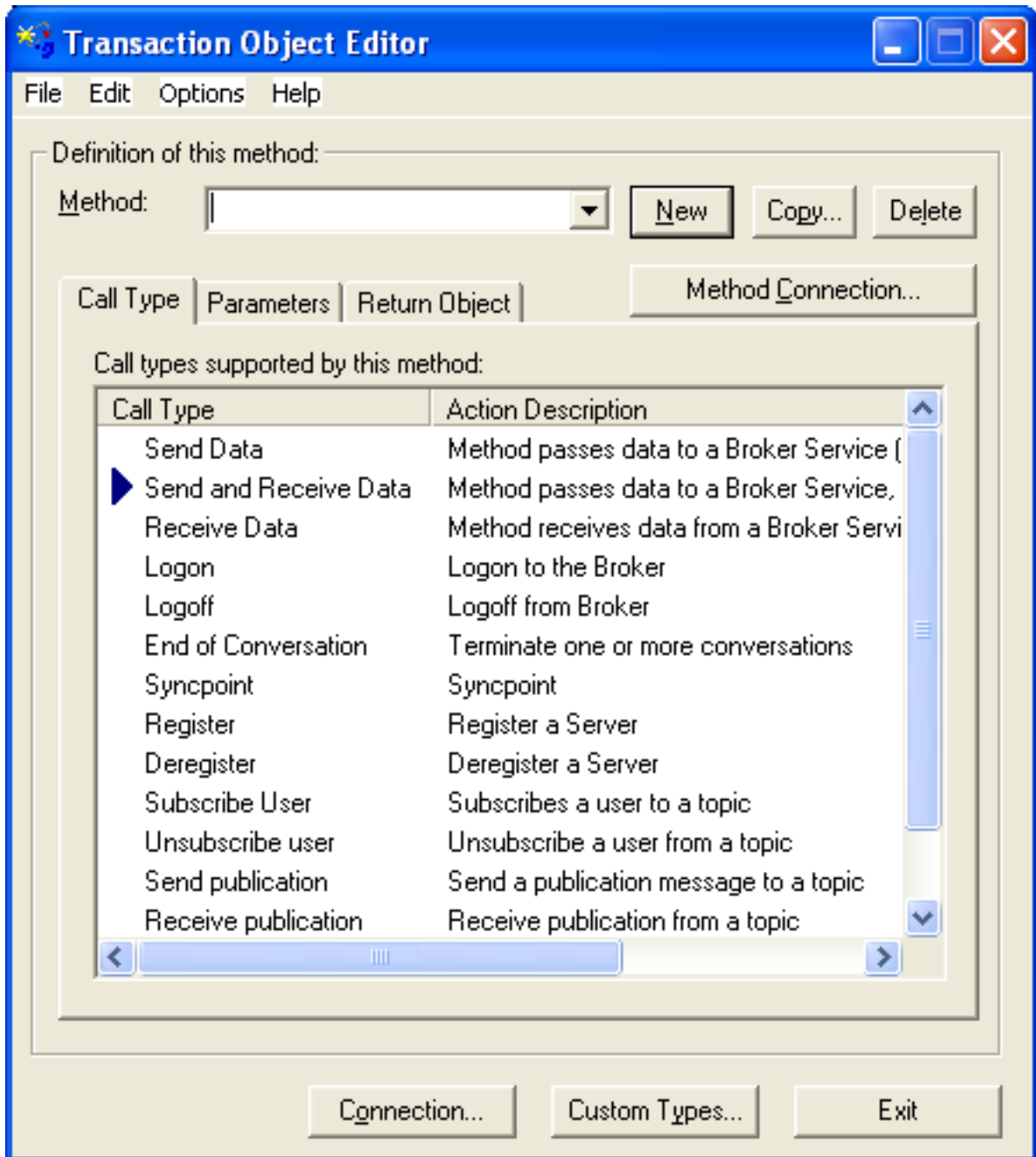
The Transaction Object Editor is a tool within Broker ActiveX Control with which you can define and maintain transaction objects. It is invoked by calling the method `DefineTOMethods` from a form that includes an ActiveX control.

The Transaction Object Editor can be called directly using the `TORedit` executable. The extension ".tor" is registered as a file type, so you can call the Transaction Object Editor with a double click from the Windows Explorer.

When the Transaction Object Editor is started, a license check is performed. If there is no license file or if the license has expired, the editor will be closed.



Note: Before you start the TOR Editor for the first time, you need to register the required DLL `ebx.dll` to your Windows system manually. Simply open a DOS prompt in folder `<drive>:\SoftwareAG\EntireX\bin` and run the command `regsvr32 ebx.dll`. If you later want to use a TOR Editor from a different installation directory, register the corresponding `ebx.dll` as above.



When a transaction object is loaded, the corresponding file name will be displayed in the title bar. If loading or saving fails, an error message will be displayed in the title bar.

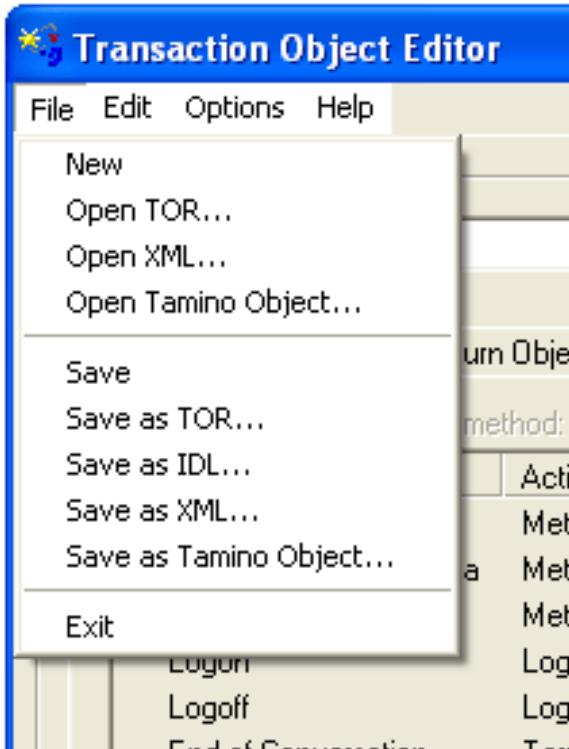
Managing TOR Files

The following functions are available for managing TOR files.

- File Menu
- Edit Menu
- Options Menu

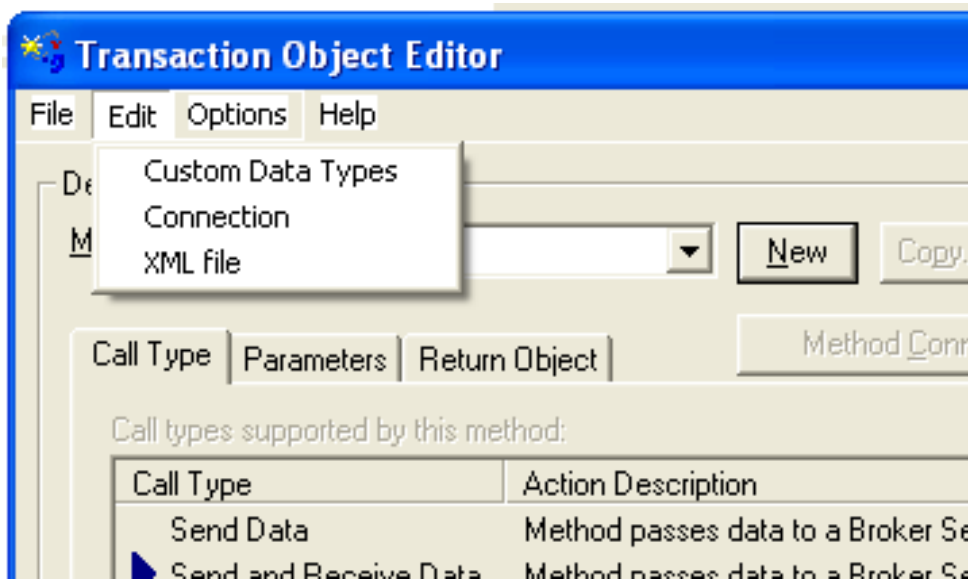
- Help Menu

File Menu



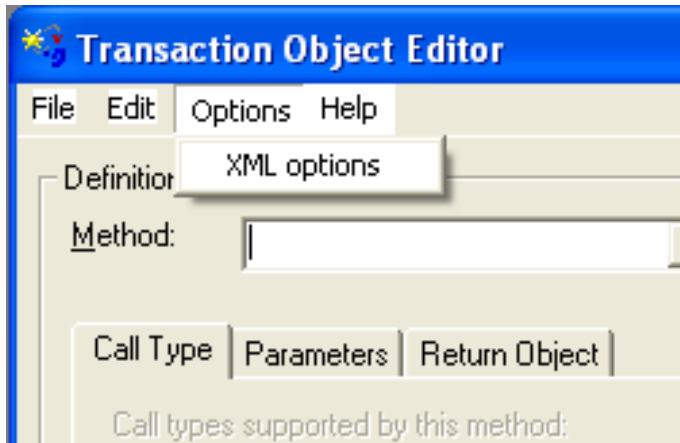
Menu Item	Description
New	Resets the TOR Editor.
Open TOR	Loads an existing TOR file. A standard Open File dialog will be displayed. This function is needed to modify an existing TOR file.
Open XML	Loads an existing XML file. A standard Open File dialog will be displayed. This function is needed to modify an existing XML file (see Loading an XML File).
Open Tamino Object	Loads an existing Tamino Object. The Open Tamino Object dialog will be displayed. This function is needed to modify an existing Tamino object (see Loading Tamino Objects).
Save	Saves a TOR file.
Save as TOR	Saves a new or modified TOR file. A standard Save File dialog will be displayed.
Save as IDL	Saves a file in IDL format. If you have made any changes to the TOR file, you must first save it in TOR file format.
Save as XML	Saves a file in XML format. A standard Save File dialog will be displayed.
Save as Tamino Object	This function saves a file in Tamino. The Save Tamino Object dialog will be displayed.
EXIT	Closes the TOR Editor.

Edit Menu



Menu Item	Description
Custom Types	Calls the Custom Data Types dialog.
Connection	Calls the Connection dialog.
XML File	Calls a standard Open File dialog. When a file is selected, a text editor will be opened.

Options Menu



Menu Item	Description
XML options	Calls an XML Options dialog.

Help Menu

Menu Item	Description
Contents	Displays the Broker ActiveX Control online help.
About	Displays the About box.

Defining Methods

The following buttons are available in the transaction method definition model:

- The **New** button causes the method name within the dialog box to be added to the store.
- The **Copy** button copies the currently selected method to a new method.
- The **Delete** button removes the selected method from the store.

Methods are logically grouped in a transaction object. Each method specified in the transaction object relates directly to a specific Broker service. To define a new method, therefore, you need to know which services are available. Each method requires the following information:

- Connection
- Call Type
- Parameters

- [Return Object](#)

Connection

Connection information is specified using the **Broker Connection Information** dialog. Each TOR file has default connection information, and each method has its individual connection information. If a parameter is not defined in the connection information of a method, the default is taken. For a description of the parameters, see [Defining Connection Information](#).

Call Type

The **Call Type** tab represents the call types that can be used for this method.

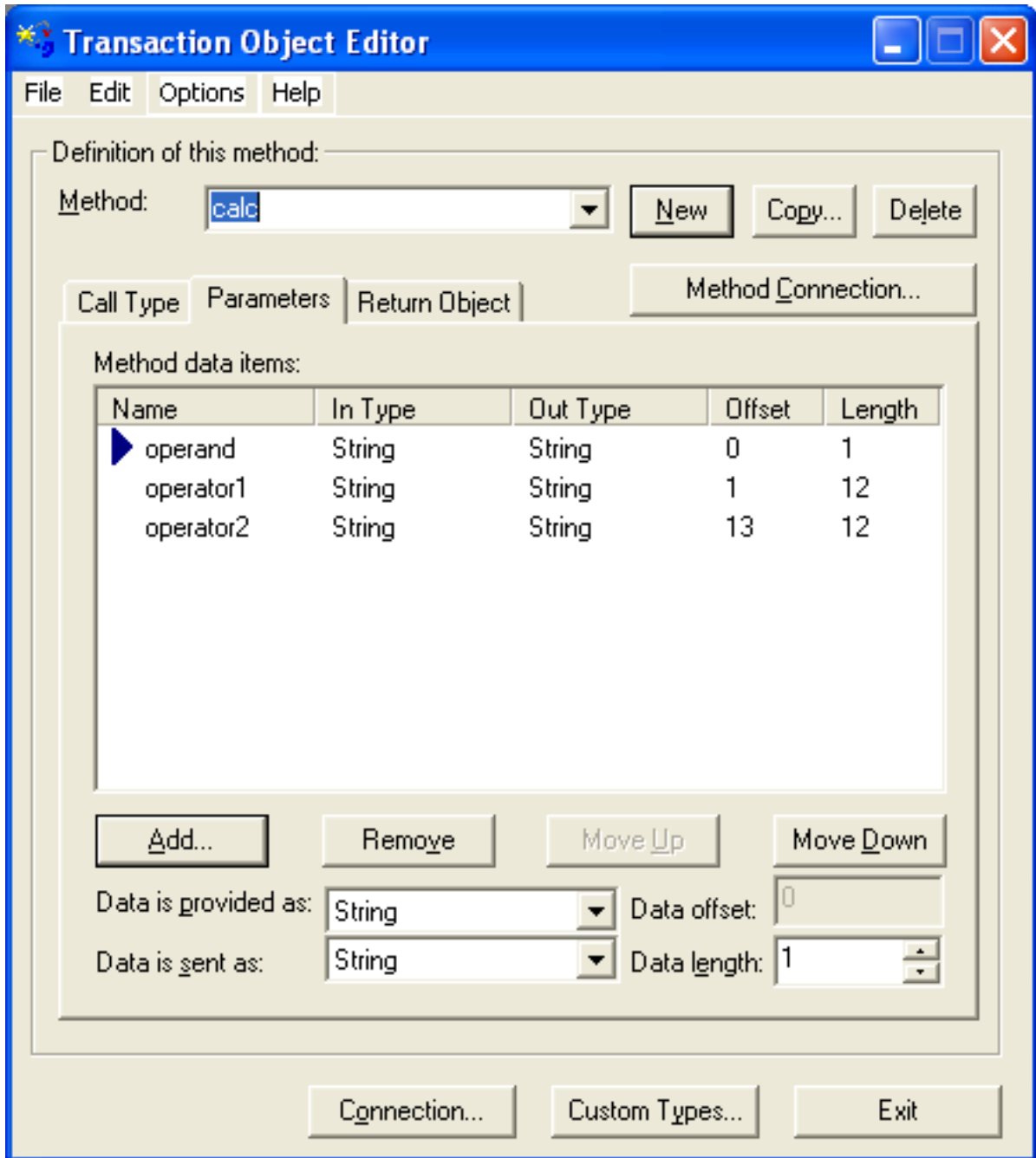
Call Type	Description
Send Data	Used to define a method that accepts parameters but does not return data from the service. This could be used to notify a back-end application of some event without waiting for a response.
Send and Receive Data	Used to define a method that accepts parameters and returns data from that service.
Receive Data	Can be used to get information from a back-end application that requires no input, for example MOTD (message of the day) information. It is also used to wait for incoming requests if you are using Broker ActiveX Control to write Broker Server applications.
Logon	Logon to EntireX Broker.
Logoff	Logoff from EntireX Broker.
End of Conversation	Used to end a conversation.
Syncpoint	Used to commit, backout, or cancel a unit of work, obtain the status of a unit of work, or delete the persistent status of a unit of work.
Register	Informs EntireX Broker that a service is available.
Deregister	Removes previously registered services from EntireX Broker's active list.
Subscribe User	Used to subscribe a user to a topic.
Unsubscribe User	Used to unsubscribe a user from a topic.
Send Publication	Used to send a publication message to a topic.
Receive Publication	Used to receive a publication message from a topic to which the user was previously subscribed.
Control Publication	Used to commit or backout a publication message.

The **Call Type** tab is shown in the [screen above](#).

Parameters

The **Parameter** tab exposes a multiline box containing individual parameter variables.

These parameters are placed into the SEND-BUFFER of the EntireX Broker call. Each parameter has a data type (Integer, Real, String etc.) and a length.



Defining a Parameter List

If data is sent, it is necessary to define a parameter list for this method. The T0 method parameter list serves as a "map" between the types passed as parameters, and the data types and locations within the method's send buffer. Items within the T0 method parameter list are ordered sequentially as they will be passed when the method is invoked.

List Control

A list control is used for defining, removing and ordering parameters of the current method. The list control supports in-place editing of items names, and works together with the item configuration controls positioned below. When a particular item is selected, it can be moved up and down the list sequentially. The order of the list defines the order in which parameters are passed when the method is invoked. Note that offsets are automatically generated for each list item, relative to the start of the list, and the items (and their sizes) that precede it.

The Add function adds the field after the selected position.

Data Conversion

Data conversion is also supported between a type provided by the client and the type expected by the Broker service. For parameters, the user can specify the data type that will be provided, and the type that will be sent to the Broker service. For return objects, the data received by the Broker service can be set to the data type retrieved by the user. The important data types are those sent to and received from a Broker service. Broker ActiveX Control automatically converts between the data type received from the Broker and a data type specified by the user (see the **Data is received as** and **Data is retrieved as** fields in the screen below).

Implemented Data Types

The scalar data types supported by the Broker ActiveX are a subset of the standard Automation VARIANT types and are listed below. In cases where the selected data type is of fixed length, the data length edit control is set to the appropriate length and grayed.

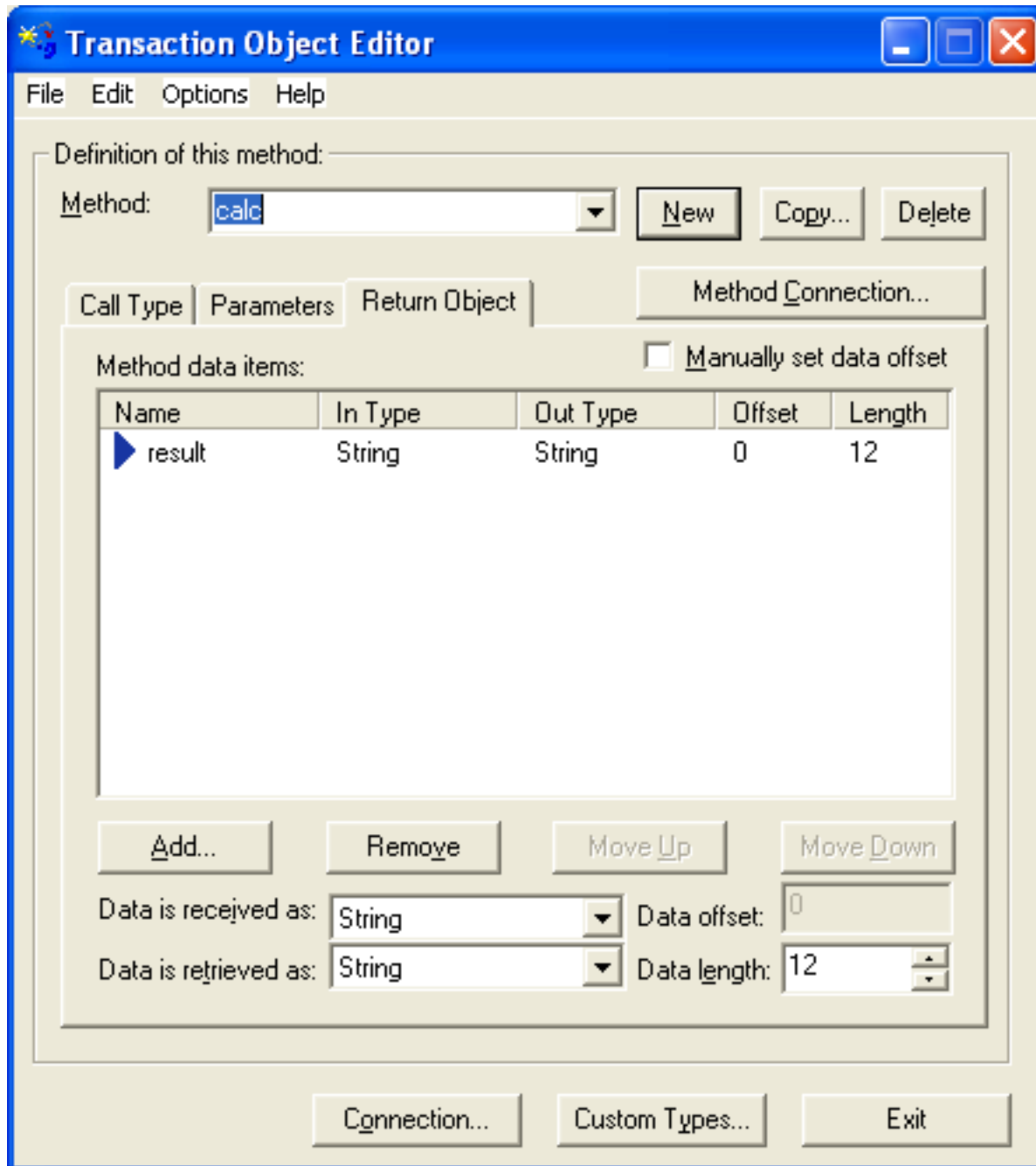
Transaction Object Method Data Types	Description
1-byte Integer	1-byte Integer used for signed and unsigned.
2-byte Integer	2-byte Integer used for signed and unsigned.
4-byte Integer	4-byte Integer used for signed and unsigned.
4-byte Real	4-byte Real compatible with "C" float.
8-byte Real	8-byte Real compatible with "C" double.
Bool	Boolean variable.
String	String of specified length.
Blob	Generic byte block.

Transaction Object Method Data Types	Description
Padding	Used to separate types in the buffer.

Return Object

If the transaction object method is invoked with call type 'Send and Receive' or 'Receive', a Return Object is created. This is a logical object that enables you to retrieve multiple scalar values or records by referencing its properties.

The **Return Object** tab exposes the individual properties that are mapped onto the RECEIVE-BUFFER of the Broker call. When the data is returned from the Broker service, Broker ActiveX Control uses the types and lengths of the defined properties to populate the values of the properties. You can now access the contents of the receive buffer as ActiveX properties of the method that is created by loading the transaction object.



As with the parameters, Broker ActiveX Control calculates the offset in the RECEIVE-BUFFER for each property. For information on list control, data conversion and implemented data types, see [Defining a Parameter List](#).

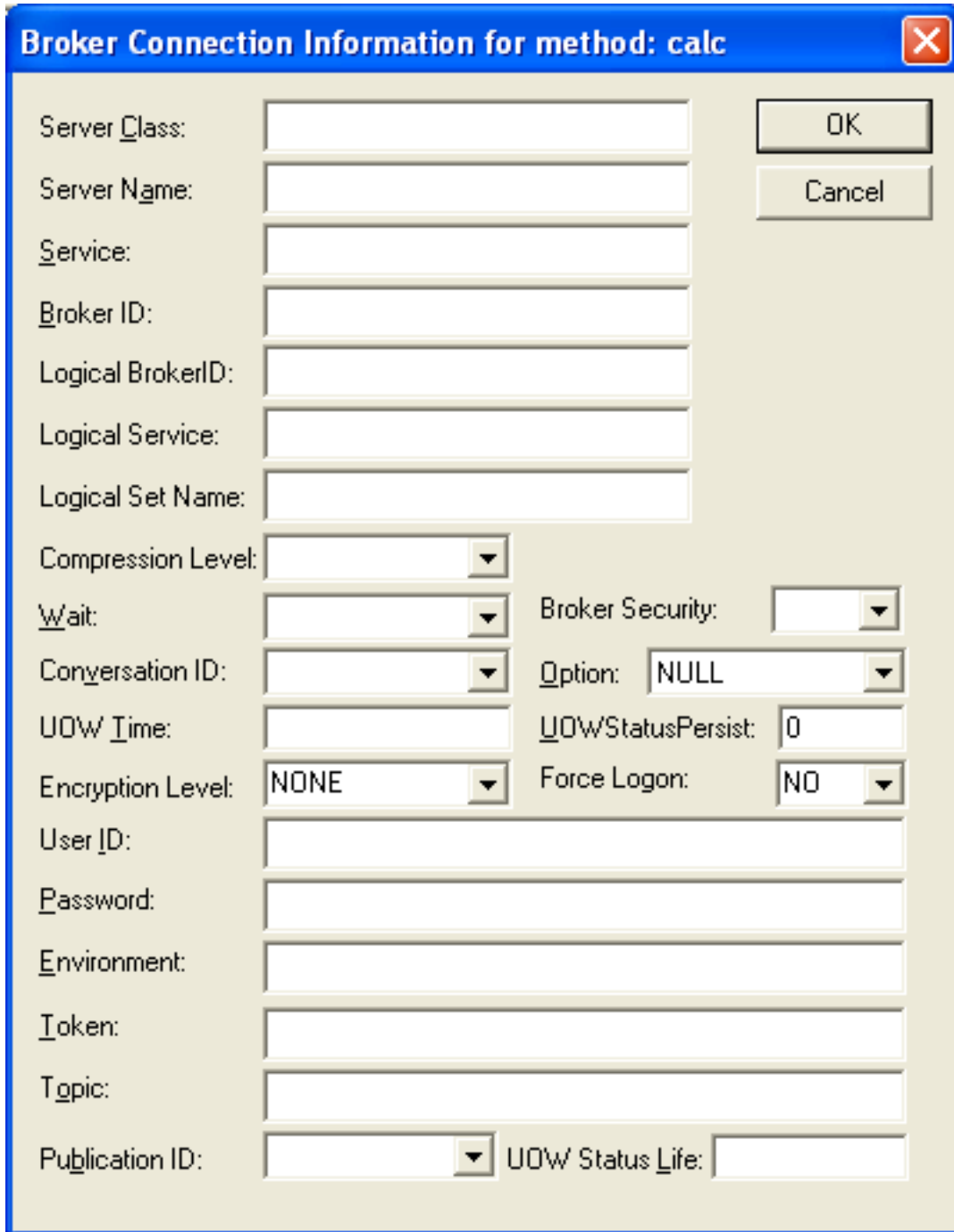
Custom Data Types are used for non-scalar data types such as arrays and structures. They are also used to assign aliases to parameters for consistent naming purposes.

The **Manually set data offset** check box allows the transaction object designer to override automatic offset calculation and specify offsets manually. This feature is powerful, but also potentially dangerous, because no base type checking can be performed.

Specifying Connection Information

Connection information relates directly to the Broker service that you want to communicate with when using this method.

Transaction methods are defined using the Transaction Object Editor. Connection information is specified using the **Broker Connection Information** dialog. Each TOR file has default connection information, and each method has its individual connection information. If a parameter is not specified in the connection information of a method, the default is taken. The Broker parameters are part of this connection information (with the exception of `Function`, which depends on the Call Type).



The image shows a dialog box titled "Broker Connection Information for method: calc". It contains various input fields and dropdown menus for configuring a broker connection. The fields are arranged in a grid-like fashion. On the right side, there are "OK" and "Cancel" buttons. The fields include:

- Server Class: [Text Box]
- Server Name: [Text Box]
- Service: [Text Box]
- Broker ID: [Text Box]
- Logical BrokerID: [Text Box]
- Logical Service: [Text Box]
- Logical Set Name: [Text Box]
- Compression Level: [Dropdown Menu]
- Wait: [Dropdown Menu]
- Conversation ID: [Dropdown Menu]
- UOW Time: [Text Box]
- Encryption Level: [Dropdown Menu, set to NONE]
- User ID: [Text Box]
- Password: [Text Box]
- Environment: [Text Box]
- Token: [Text Box]
- Topic: [Text Box]
- Publication ID: [Dropdown Menu]
- Broker Security: [Dropdown Menu]
- Option: [Dropdown Menu, set to NULL]
- UOWStatusPersist: [Text Box, set to 0]
- Force Logon: [Dropdown Menu, set to NO]
- UOW Status Life: [Text Box]

The **Broker Connection Information** dialog box accepts all the parameters required for establishing the necessary Broker connection to execute the defined method/call type.

Connection Information Parameters

Parameter	Description								
BrokerID	The unique name of the Broker node that the services are attached to. Information in this dialog can be changed without affecting the application code. For example, if the <code>BrokerID</code> changed, you would change the connection information in the methods (services) affected and distribute the new transaction object file. The next time the application code loads the transaction object file and calls a method, the new connection information will be used.								
CompressLevel	Compression level. Valid values: N Y 0-9. See also <i>Data Compression in EntireX Broker</i> in the general administration documentation.								
ServerClass, ServerName, Service	These three parameters represent the unique “signature” of this method call.								
Wait	The following values are set for this parameter, depending on the operation: <table border="0" style="margin-left: 20px;"> <tr> <td>Operation</td> <td>Wait Value (in seconds)</td> </tr> <tr> <td>Send</td> <td>0</td> </tr> <tr> <td>Send and Receive</td> <td>30 (*)</td> </tr> <tr> <td>Receive</td> <td>59 (*)</td> </tr> </table> (*) if no value is specified in the Connection info.	Operation	Wait Value (in seconds)	Send	0	Send and Receive	30 (*)	Receive	59 (*)
Operation	Wait Value (in seconds)								
Send	0								
Send and Receive	30 (*)								
Receive	59 (*)								

See [Properties of Broker ActiveX Control](#) for a description of the other parameters.

Setting the Broker Call Parameters

Calling a method of a transaction object results in a Broker call. The parameters for the Broker call are taken either

- from the **Broker Connection Information** dialog, see above, or
- from the properties (see [Properties of Broker ActiveX Control](#)).

If a value is specified in the **Connection Information** dialog, this value is taken and overrides any value specified in the properties.

If no value is specified in the **Connection Information** dialog, the current setting of the properties is taken. Leaving these parameters blank in the **Connection Information** dialog enables you to change these parameters dynamically, and also enables Broker communication in conversational mode. See example below:

Visual Basic Example

This example shows a possible usage of dynamic parameter assignment:

```
Set TransObject=BOCX.CreateTransObject ("...calc.tor")
BOCX.UserID = "USER1"
BOCX.BrokerID = "ETB121"
Set ReturnOb = TransObject.calc("+", "000000000001", "000000000002")
```

Defining Custom Data Types

The **Custom Data Types** dialog allows you to define new data types that will appear in the **Return Object** tag. With the **Apply** button you can embed a custom type within another custom type as long as this does not result in a recursive inclusion.

The following four classes of custom data types are supported:

- Custom Data Type 'Alias'
- Custom Data Type 'Array '
- Custom Data Type 'Record'
- Custom Data Type 'Structure'

Any custom data type can be used in transaction objects return objects. Custom data types are not supported as method parameters.

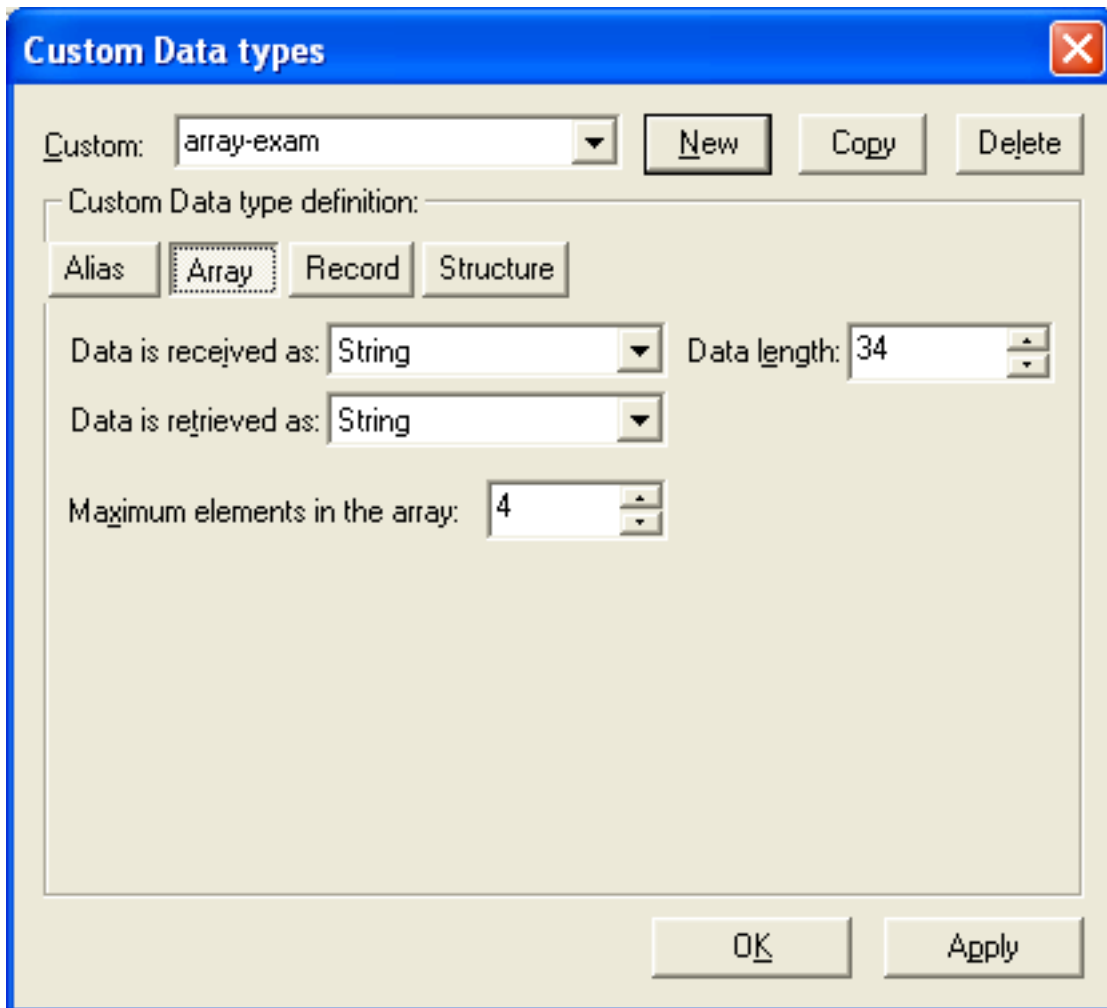


Note: All custom data types can be used recursively. That is, any custom data type can be used as a member or base type for any other custom type. This allows for nested structures, as well as arrays within structures and records.

Custom Data Type 'Alias'

An *alias* is a custom data type that allows an administrator to specify an alias for any defined data type - custom or not. Aliasing also allows the definition of data types with specific in and out data types (type translation).


Custom Data Type 'Array'



The screenshot shows a dialog box titled "Custom Data types" with a blue header and a red close button. The "Custom:" dropdown menu is set to "array-exam". To the right are "New", "Copy", and "Delete" buttons. Below this is a section for "Custom Data type definition:" with three tabs: "Alias", "Array" (which is selected), and "Structure". Under the "Array" tab, there are three fields: "Data is received as:" set to "String", "Data length:" set to "34", and "Data is retrieved as:" set to "String". Below these is a field for "Maximum elements in the array:" set to "4". At the bottom of the dialog are "OK" and "Apply" buttons.

An *array* consists of multiple serial elements of the same data types. Arrays can be made up of either scalar or custom data types. The number of elements in an array must be specified.

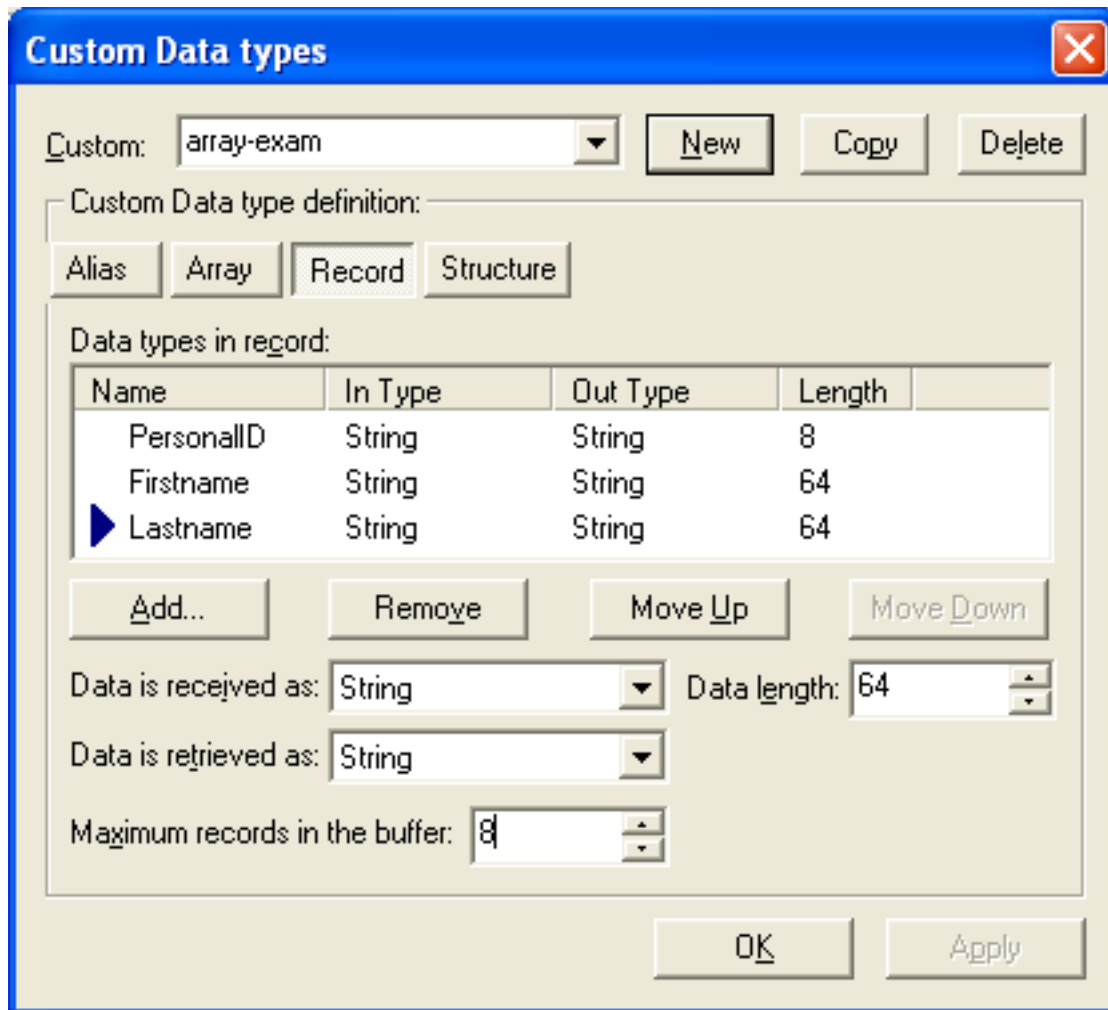
Array custom data types accept the same basic information as alias data types, with the addition of the number of elements in the array. Arrays allow elements of the specified base type to be accessed in a subscripted fashion.

 **Note:** Multidimensional arrays and arrays of structures can be implemented by specifying a custom array or record data type as the base type of this array.

Custom Data Type 'Record'

A *record* is a repeating collection of data types - scalar or custom.

This custom data type allows you to define a collection of data types that can be accessed in a subscripted fashion. The order of defined types in the **Record** can be changed. Also, the number of records within the receive buffer can be specified if known.



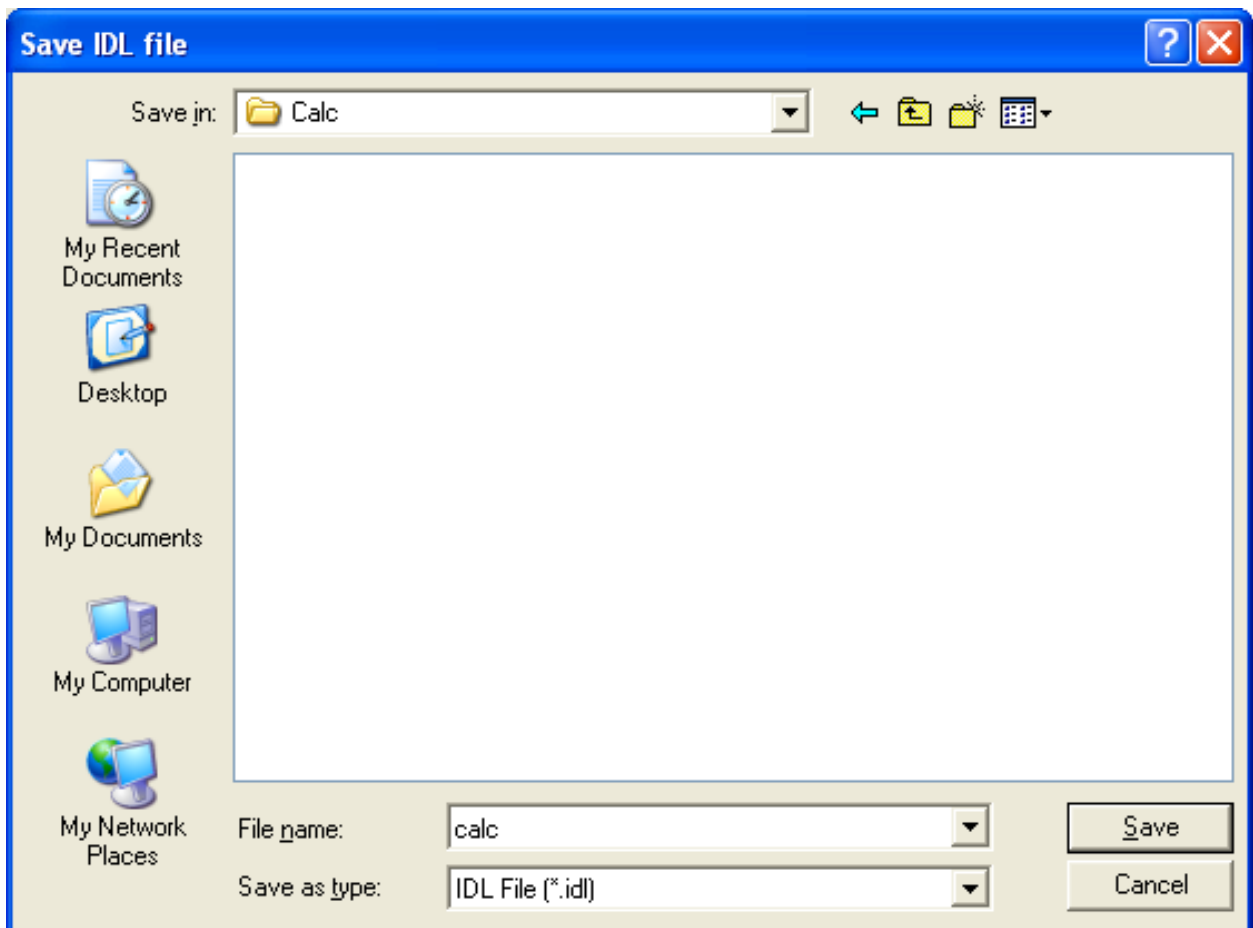
Custom Data Type 'Structure'

A *structure* is a named collection of data types.

The controls for this custom data type are identical to those of the data type 'record', with the exception of a repetitive count, which is not applicable.

TOR Files in IDL Format

When a TOR file is saved in IDL format, a file with extension `.idl` is generated. (The file must have been saved as a TOR file before).



This IDL file can be used by other EntireX tools such as DCOM Wrapper or Java Wrapper. It can be modified with any editor like a regular IDL file.

Conversion Rules

List of the performed conversions:

In TOR file	Converted to ... in IDL file
TOR file name	Library name
Methodname	Program name
Connection Info	"Server address" as comment
DataItems in Parameter Map	"In" Parameters
DataItems in Return Map	"Out" Parameters
Manual Offsets in Return Map	Will not be converted. If "manual offsets" is marked in a method, a comment is generated for this program.
Custom Data Types	The names of the CDTs used are displayed in a comment.
- Alias	Nothing
- Array	A dimension specification
- Record	A dimension specification and a group
- Structure	A group
Format Conversion	The IN-Type of the Parameter Map and the OUT-Type of the Return Map are used.
- I1	I1
- I2	I2
- I4	I4
- Real 4	F4
- Real 8	F8
- Bool	L
- String	A<size>
- Blob	B<size>
- Padding	B<size>

TOR Files in XML Format

To use TOR files in XML format, Internet Explorer 5 or above is required.

Loading an XML File

When you load an XML file, the XML file is checked against the defined DTD (see [The DTD File](#) list below). When you use the XML file, it is not necessary to store the transaction object in TOR file format.

Saving an XML File

When a TOR file is saved in XML format, a file with the extension .xml is generated.

This XML file can be viewed with a browser that supports XML. It can also be viewed and edited with any XML notepad or any text editor.

The DTD File

The structure of the XML file is defined in the DTD file. When you use a tool that validates XML files, the XML file is checked against these definitions.

Entry in the DTD file	Explanation	
<code><!ELEMENT EntireXTorFile (DefaultConnection? , Method*, CDT*)></code>	The root must always be defined. It contains: <ul style="list-style-type: none"> ■ 0-1 default connections ■ 0-n methods ■ 0-n CDT (= custom data types) 	
<code><!ATTLIST EntireXTorFile Name CDATA #IMPLIED Version CDATA #IMPLIED></code>	Name	The name of the TOR file.
	Version	The EntireX version with which the XML file was generated
<code><!ELEMENT DefaultConnection EMPTY></code>	The global connection information is stored here.	
<code><!ATTLIST DefaultConnection %Connection;></code>	All parameters in the default connection are stored as attributes. See the detailed description of the %Connection at the end of this table.	

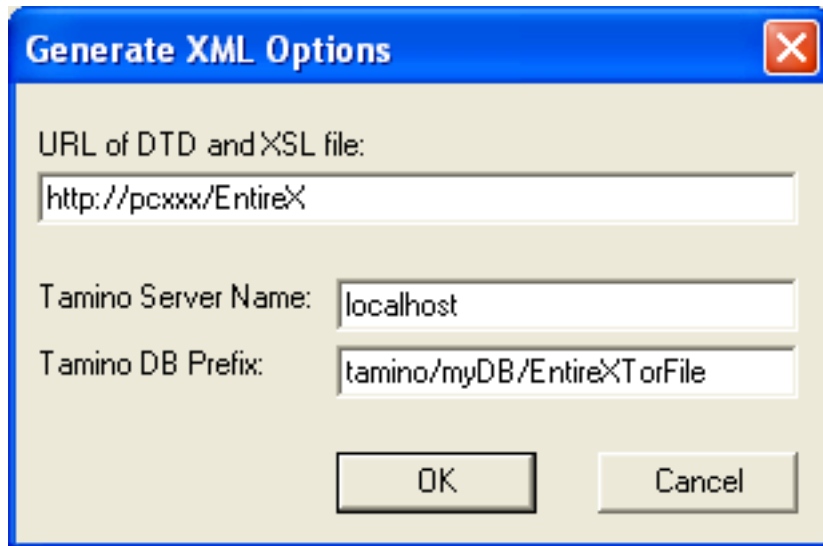
Entry in the DTD file	Explanation						
<pre><!ELEMENT Method (MethodConnection? , ← Parameter*)></pre>	<p>Each method contains:</p> <ul style="list-style-type: none"> ■ 0-1 method connections ■ 0-n parameter 						
<pre><!ATTLIST Method Name CDATA #REQUIRED CallType (SEND RECEIVE SEND-RECEIVE LOGON LOGOFF EOC SYNCPOINT REGISTER Deregister SUBSCRIBE UNSUBSCRIBE SEND_PUB RECEIVE_PUB CONTROL_PUB) #REQUIRED ManualOffset (YES NO) #IMPLIED></pre>	<p>A name and a call type must be defined for each method. The manual offset contains the manual offset switch of the return map.</p>						
<pre><!ELEMENT MethodConnection EMPTY></pre>	<p>The connection information of each method is stored here.</p>						
<pre><!ATTLIST MethodConnection %Connection;></pre>	<p>All parameters belonging to the method connection are stored as attributes. See the detailed description of the %Connection at the end of this table.</p>						
<pre><!ELEMENT Parameter (InFormat, OutFormat, ← Length?)></pre>	<p>Each parameter contains:</p> <ul style="list-style-type: none"> ■ 1 in format ■ 1 out format ■ 0-1 length 						
<pre><!ATTLIST Parameter Name CDATA #IMPLIED Direction (IN OUT INOUT) ← #IMPLIED Offset CDATA #IMPLIED></pre>	<table border="1"> <tr> <td data-bbox="876 1148 980 1190">Name</td> <td data-bbox="980 1148 1385 1190">Name of the parameter</td> </tr> <tr> <td data-bbox="876 1190 980 1308">Direction</td> <td data-bbox="980 1190 1385 1308">IN: if parameter is from the parameter map OUT: if it is from the return map</td> </tr> <tr> <td data-bbox="876 1308 980 1392">Offset</td> <td data-bbox="980 1308 1385 1392">Offset value of the return map, if ManualOffset = YES</td> </tr> </table>	Name	Name of the parameter	Direction	IN: if parameter is from the parameter map OUT: if it is from the return map	Offset	Offset value of the return map, if ManualOffset = YES
Name	Name of the parameter						
Direction	IN: if parameter is from the parameter map OUT: if it is from the return map						
Offset	Offset value of the return map, if ManualOffset = YES						
<pre><!ELEMENT CDT (Alias Array Record ← Structure) ></pre>	<p>A custom data type (CDT) is an alias, an array, a record or a structure.</p>						
<pre><!ATTLIST CDT Name ID #REQUIRED></pre>	<p>The name of the CDT is required.</p>						
<pre><!ELEMENT Alias (InFormat, OutFormat, Length?)></pre>	<p>An alias contains:</p> <ul style="list-style-type: none"> ■ 1 in format ■ 1 out format ■ 0-1 length 						

Entry in the DTD file	Explanation
<code><!ELEMENT Array (InFormat, OutFormat, Length?)></code>	An array contains: <ul style="list-style-type: none"> ■ 1 in format ■ 1 out format ■ 0-1 length
<code><!ATTLIST Array NumberEle CDATA #IMPLIED></code>	The numbers of elements for an array are stored here.
<code><!ELEMENT Record (Parameter*)></code>	The record contains: <ul style="list-style-type: none"> ■ 0-n parameter
<code><!ATTLIST Record NumberEle CDATA #IMPLIED></code>	The numbers of elements for a record are stored here.
<code><!ELEMENT Structure (Parameter*) ></code>	The structure contains: <ul style="list-style-type: none"> ■ 0-n parameter
<code><!ELEMENT InFormat (Scalar UsedCDT)></code>	An InFormat is a scalar value or a reference to a CDT.
<code><!ELEMENT Scalar EMPTY></code>	
<code><!ATTLIST Scalar Format (I1 I2 I4 F4 F8 Bool String Blob Padding) #REQUIRED></code>	A scalar must be in one of the listed formats.
<code><!ELEMENT UsedCDT EMPTY></code>	
<code><!ATTLIST UsedCDT Target IDREF #REQUIRED></code>	A UsedCDT must reference the name of a defined CDT.
<code><!ELEMENT OutFormat (Scalar UsedCDT)></code>	An OutFormat is a scalar value or a reference to a CDT.
<code><!ELEMENT Length EMPTY></code>	
<code><!ATTLIST Length Value CDATA #IMPLIED></code>	A length must be defined for scalars with the values: string, BLOB and padding or UsedCDTs.
<pre> <!ENTITY % Connection 'ServerClass CDATA #IMPLIED ServerName CDATA #IMPLIED Service CDATA #IMPLIED ConversationID (NONE NEW OLD ANY) #IMPLIED UOWTime CDATA #IMPLIED BrokerID CDATA #IMPLIED UserID CDATA #IMPLIED Password CDATA #IMPLIED </pre>	All connection parameters are defined as attributes.

Entry in the DTD file	Explanation
<pre> Environment CDATA #IMPLIED Wait CDATA #IMPLIED UOWStatusPersist CDATA #IMPLIED Option (NULL MSG HOLD IMMED ← QUIESCE EOC CANCEL LAST NEXT PREVIEW COMMIT BACKOUT SYNC ATTACH DELETE EOCCANCEL QUERY SETUSTATUS ANY TERMINATE DURABLE CHECKSERVICE) #IMPLIED Encryption (NONE TO-BROKER TO-TARGET) #IMPLIED ForceLogon (NO YES) #IMPLIED CompressLevel CDATA #IMPLIED Token CDATA #IMPLIED Topic CDATA #IMPLIED PublicationID CDATA #IMPLIED UOWStatusLife CDATA #IMPLIED BrokerSecurity CDATA #IMPLIED" > </pre>	

Defining the Location of the DTD and XSL File

A DTD file is used to check the XML file. An XSL file is used to view the XML file. To locate these files, enter a reference in the **XML Options**:



This reference can be a URL (like above) or a regular path (e.g., the default: the EntireX *etc* directory).

Using the XML Objects During Runtime

The XML file can also be used during runtime. It must be defined in the same way as the TOR file.

Visual Basic Example

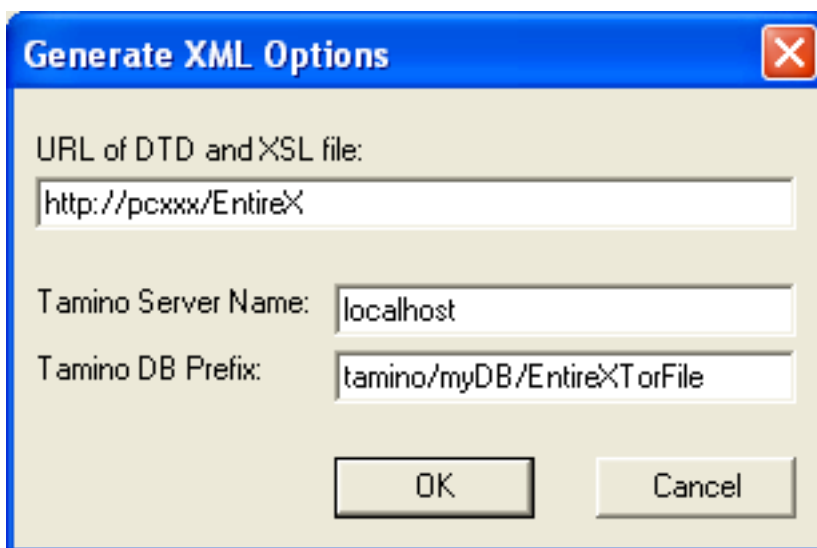
```
Set TransObject=B0CX.CreateTransObject ("...\calc.xml")
```

Storing TOR Files in a Tamino Database

To store and use TOR files in a Tamino database, Tamino 4.2.1 or higher and Internet Explorer 5 or higher are required.

Creating a Tamino Database for the TOR Files

In the EntireX *etc* directory an *EntireXTorIno vrs.xml* is provided. This file can be used to define the schema in Tamino (`_define` function). It is very close to the DTD file. The XML files generated can be directly stored in Tamino. The database prefix defined in Tamino must be defined in the **XML Options** screen as well as the server name of the Tamino database.



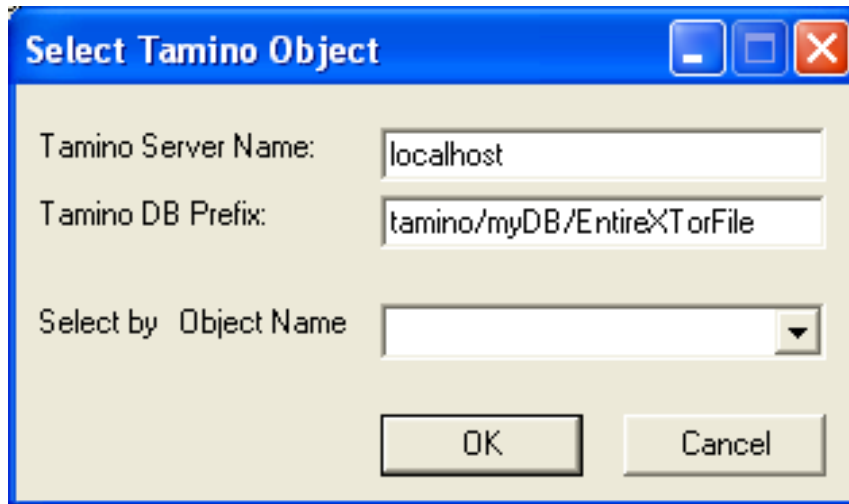
The screenshot shows a dialog box titled "Generate XML Options". It has a blue title bar with a close button (X) on the right. The dialog contains three input fields:

- URL of DTD and XSL file:
- Tamino Server Name:
- Tamino DB Prefix:

At the bottom of the dialog are two buttons: "OK" and "Cancel".

Loading Tamino Objects using the TOR Editor

When loading a Tamino object, the following dialog will be displayed:



If necessary, the Tamino server name and the Tamino database prefix can be changed here. The name of the desired object can be entered directly or selected from the drop-down menu **Select by Object Name**.

Storing Tamino Objects using the TOR Editor

When saving a Tamino object, the following dialog will be displayed:



If necessary, the Tamino server name and the Tamino DB prefix can be changed here. The name of the object must be entered in the **Tamino Object Name** field. If a Tamino object with this name already exists, you can overwrite the existing file or cancel the save operation.

Using Tamino Objects During Runtime

The Tamino object can also be used during runtime. It must be defined like the XML file:

Visual Basic Example

```
Set TransObject=BOCX.CreateTransObject ("Calc")
```



Note: The name of the Tamino object is case-sensitive.

The Tamino server name and the Tamino DB prefix from the **General XML Options** screen are used.

7 Calling Broker ActiveX Control Remotely

- Setting up the Server Environment 60
- Setting up the Client Environment 65
- Testing the Connection 68

You can call Broker ActiveX Control remotely if you use it as an automation server. This means you can use the Broker component from a separate process - either on the same machine or on another machine in the network.

Setting up the Server Environment

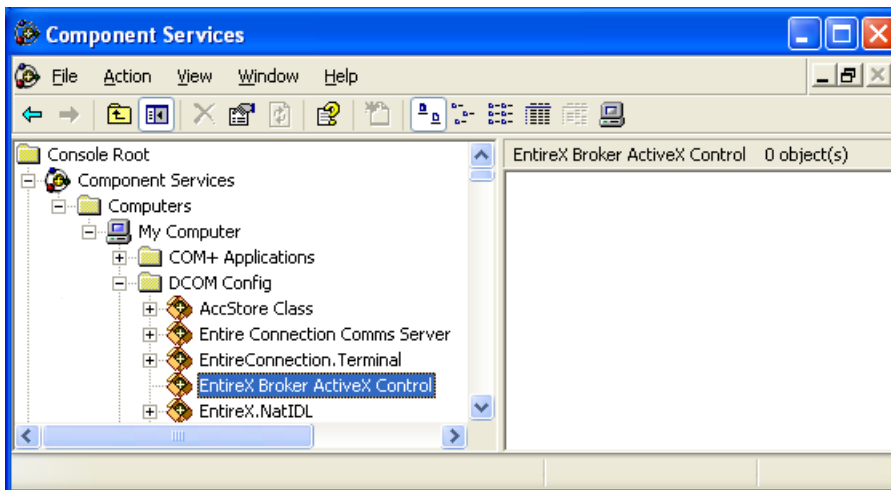
To configure the security settings use **Component Services** from the **Administrative Tools** in the **Control Panel**.

Below is a step-by-step guide on how to configure the server environment:

Step 1

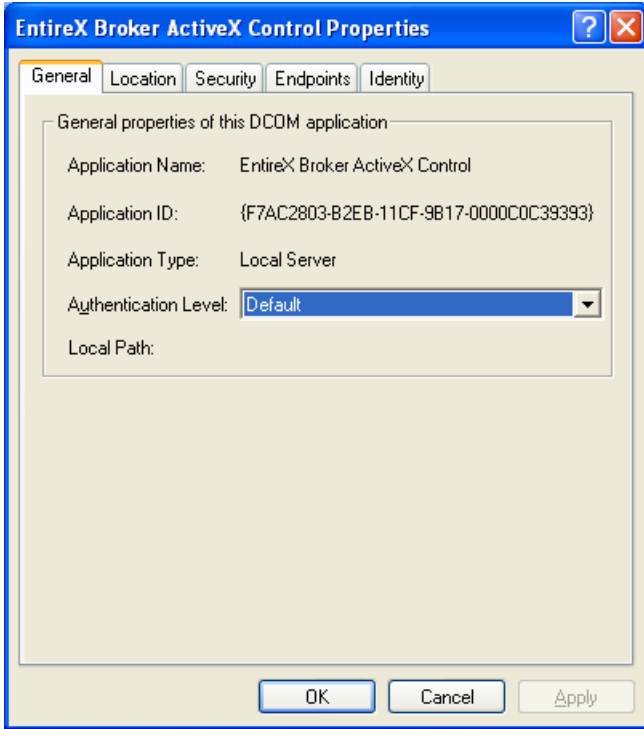
Open the **Component Services** on the server.

The following dialog box will be displayed:



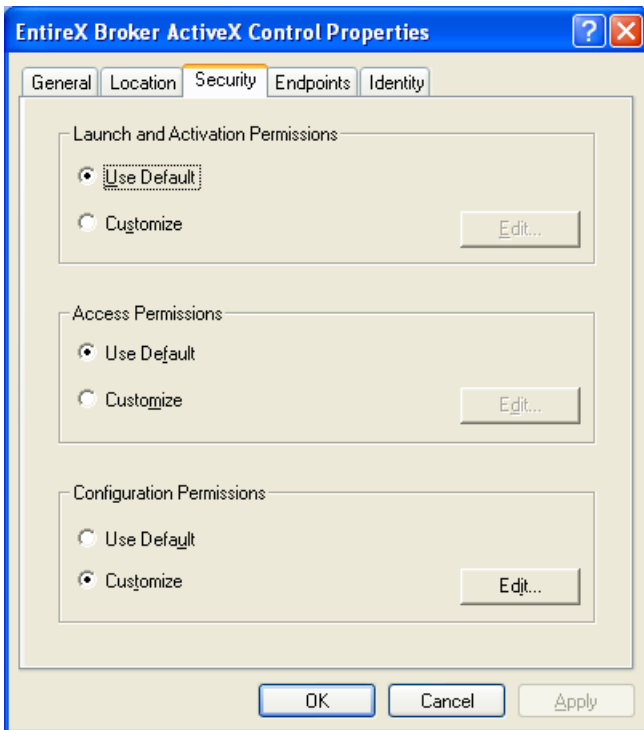
Select **EntireX Broker ActiveX Control** in the **DCOM Config** list box and choose the properties from the context menu.

The following dialog box will be displayed:



Step 2

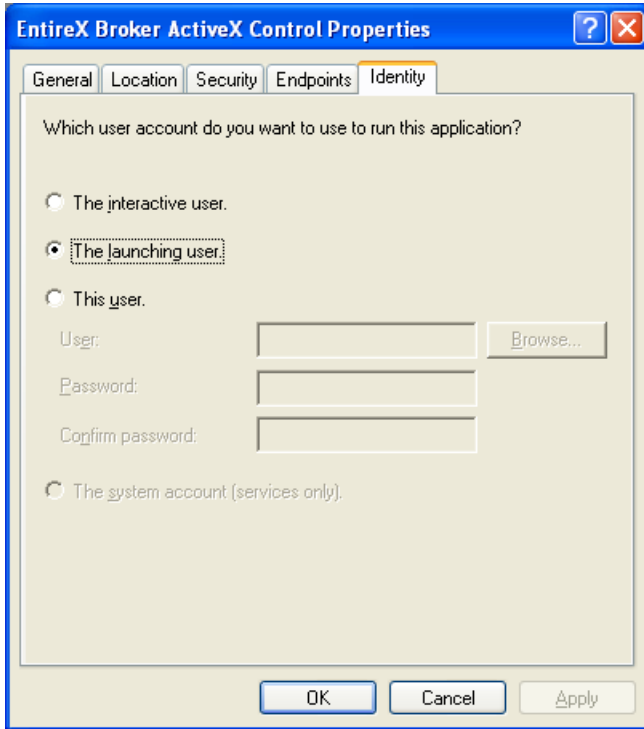
Click the **Security** tab.



In the dialog box displayed above, keep the defaults for access, launch and configuration permissions.

Step 3

Click the **Identity** tab.



There are three options to define the user account to be used to run the application:

- **The interactive user**

This implies that a user with permission to launch the application must be logged on to the server machine.

- **The launching user**

This implies that an account must be created on the server machine with the same username/password as on the client machine. This account will then be used to launch the application.

- **This user**

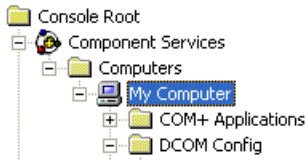
A final option is to specify a user account to be used when launching the application.

In each case, the username/password of the client machine must also exist on the server machine.

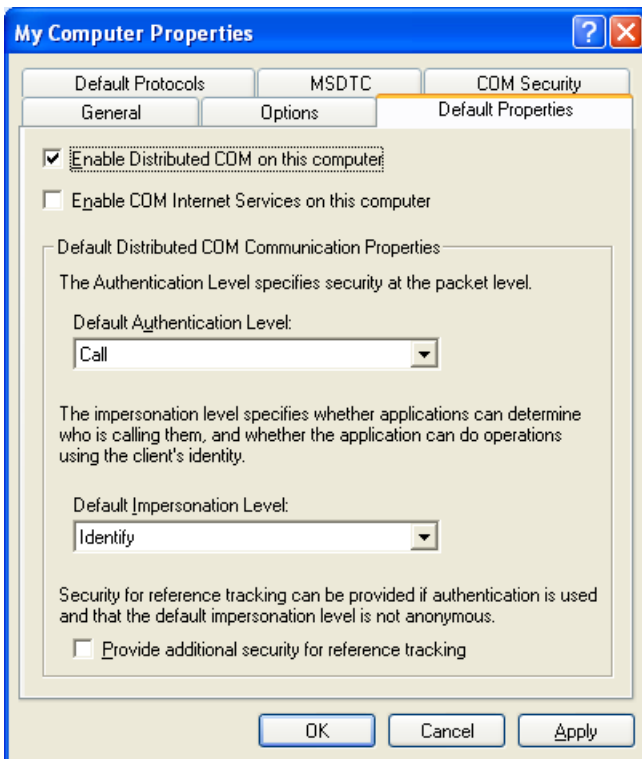
Select one of the options and choose **OK** to return to the **Component Services**.

Step 4

Click on **My Computer** and choose the properties from the context menu.



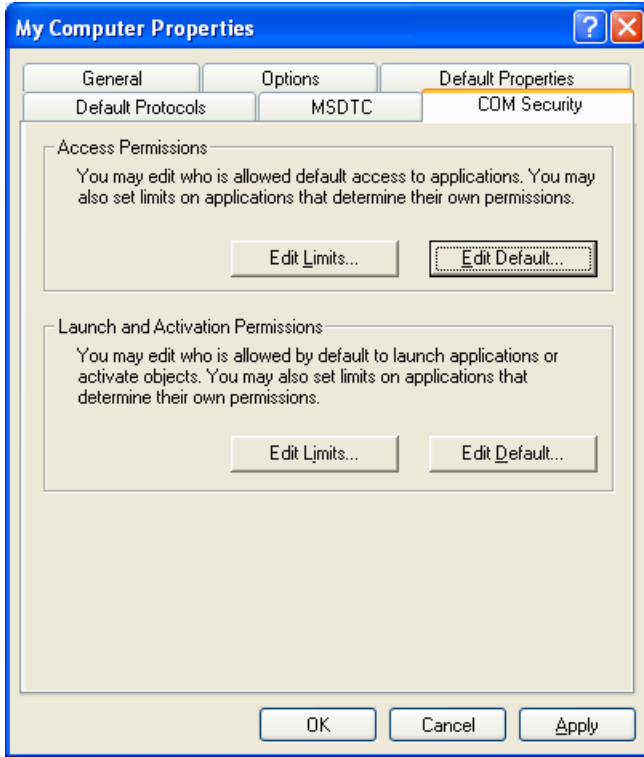
The following dialog box will be displayed; click on the **Default Properties** tab.



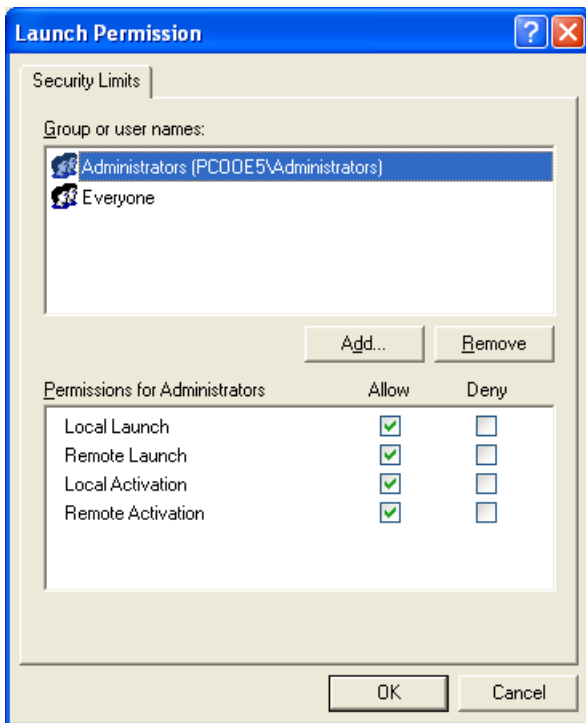
Choose the options as shown in the dialog box above.

Step 5

Click on the **COM Security** tab.



In the **Launch and Activation Permissions** area of the dialog box displayed above, choose **Edit Default**. The following dialog box will be displayed:



Make sure that either the user corresponding to the client machine account, or a group to which the user belongs, has **Allow Launch** as **Type of Access**.

Choose **OK** in this screen and then **Apply**, and exit **Component Services** on the server.

Setting up the Client Environment

The *EbxProxy.dll* is installed by default on the server in directory `<drive>:\SoftwareAG\EntireX\bin`. Copy the file from the server machine to the client machine.

The DLL must then be registered with: `REGSVR32 <path>\EBXproxy.dll`.

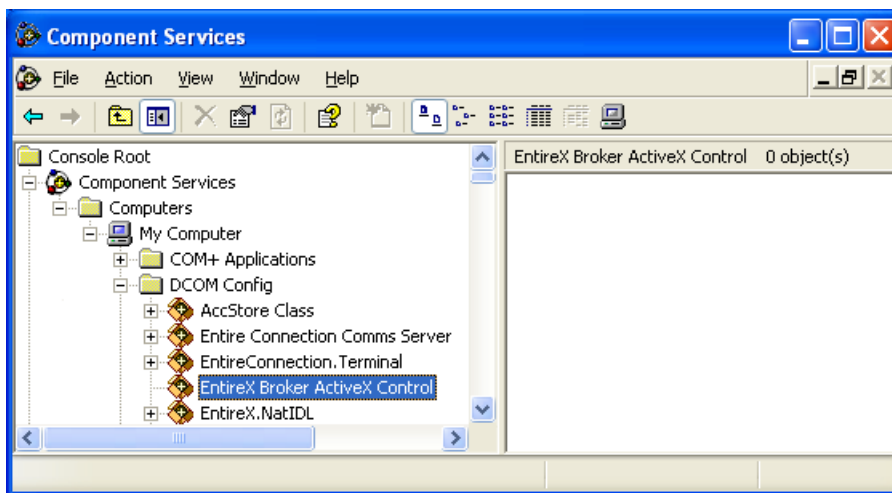
To configure the client environment use **Component Services** from the **Administrative Tools** in the **Control Panel**.

Below is a step-by-step guide on how to configure the client environment:

Step 1

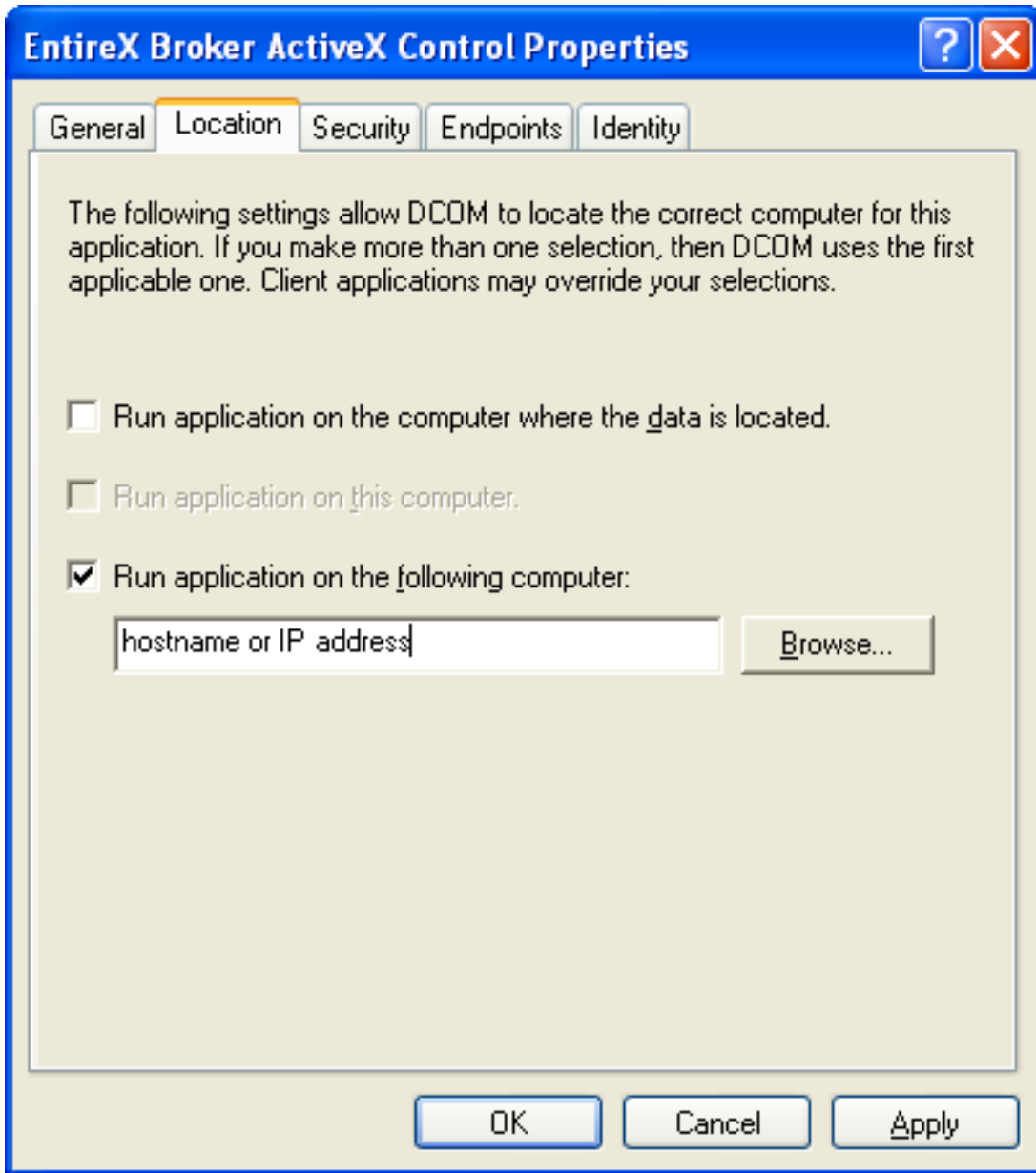
Open the **Component Services** on the client.

The following dialog box will be displayed:



Select **EntireX Broker ActiveX Control** in the **DCOM Config** list box, choose the properties from the context menu and click the **Location** tab.

Step 2

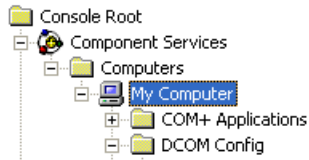


In the **Location** tab of the **ActiveX Control Properties** dialog box above, select the check box **Run application on the following computer:** and enter either the hostname or the IP address of the server machine.

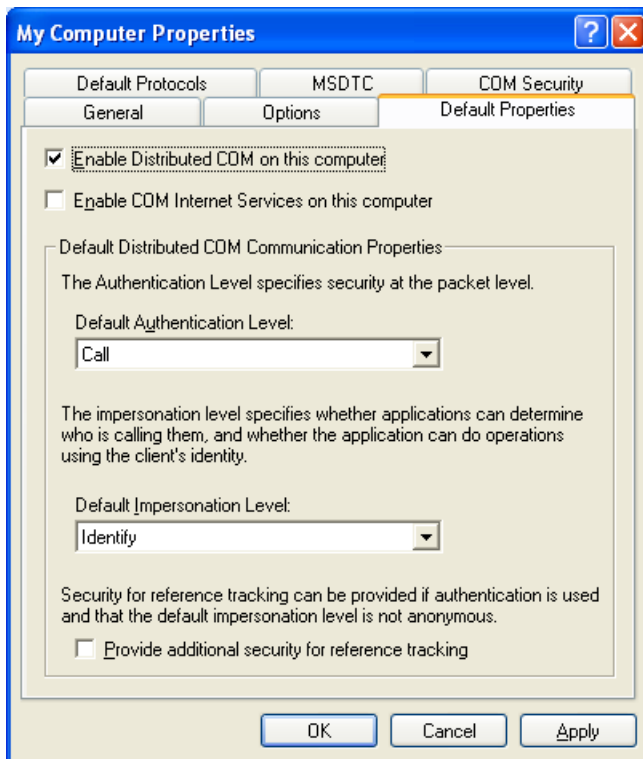
Choose **Apply** and then **OK**.

Step 3

Select **My Computer** and choose the properties from the context menu.



The **My Computer Properties** dialog box will be displayed. Select the **Default Properties** tab.



Choose the check box **Enable Distributed COM on this computer**, set the default authentication level to **Call** and the default impersonation level to **Identify**.

Choose **OK**.

Testing the Connection

You are now ready to test the connection between the client machine and the server machine.

Test the TCP/IP Connection

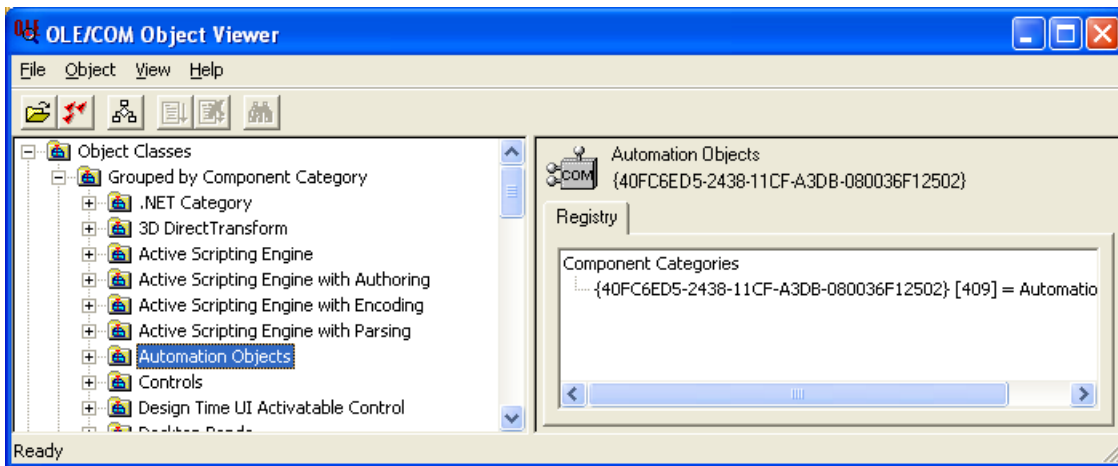
Test the TCP/IP connection between the client and the server (use, for example, PING).

Test the Remote Call

To test whether an application can be called remotely, you can use the OLE/COM Object Viewer:

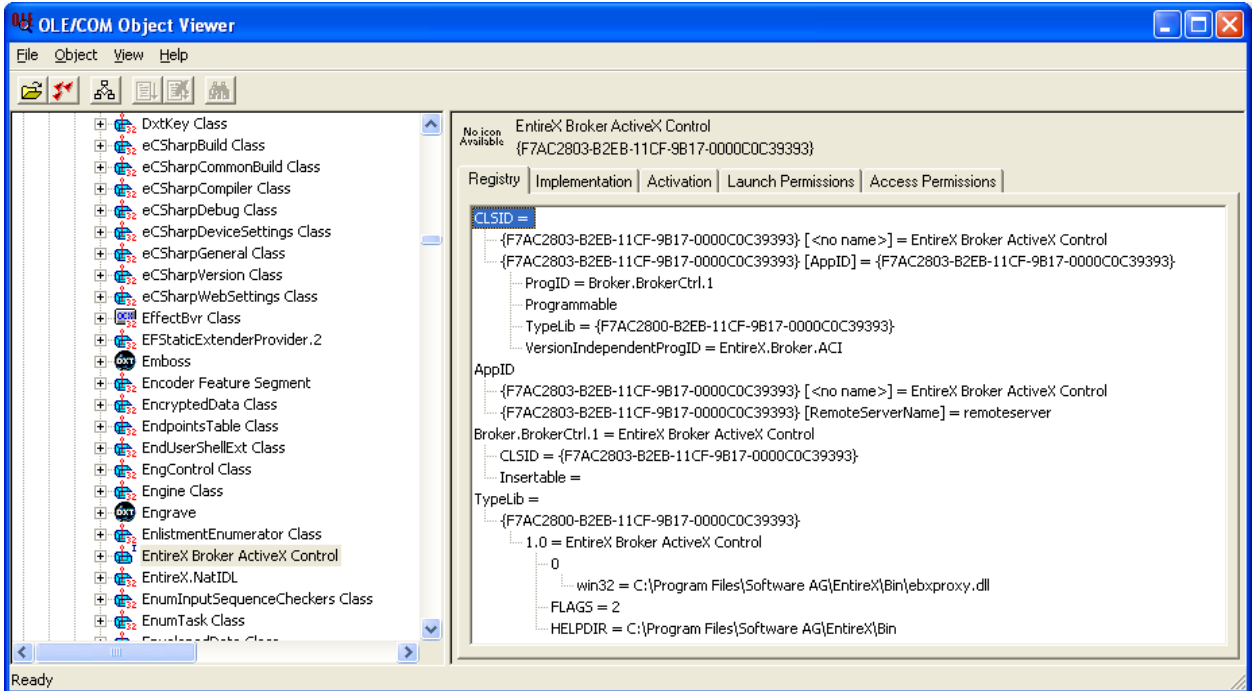
Run the OLE/COM Object Viewer on the client.

The **OLE/COM Object Viewer** dialog box will be displayed:



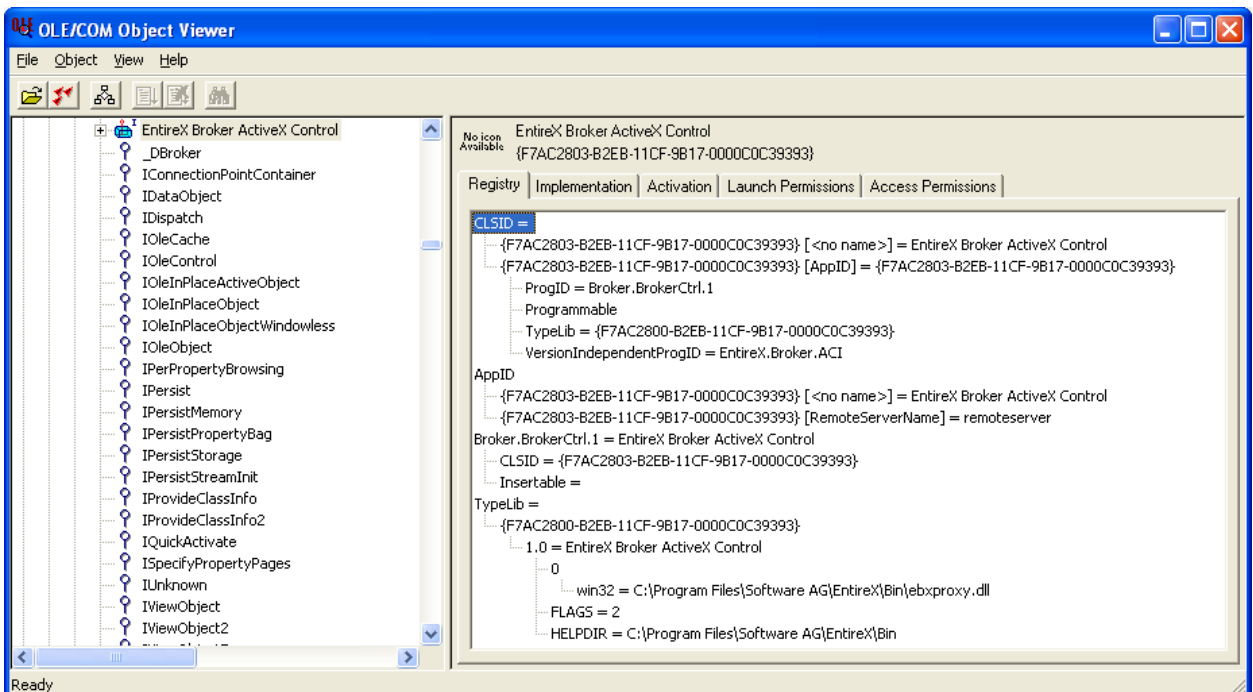
Select **Automation Objects** in the navigation frame to display a list of all the automation objects on the client machine.

A screen similar to the one displayed below will be displayed:



Select **EntireX Broker ActiveX Control**, open its context menu and choose **Create Instance**.

If the remote call is successful, the EntireX Broker component on the server machine will be called and the following screen will be displayed:



If you receive an error message (for example “Class not registered”) please check the following:

- the TCP/IP connection (with PING)
- the security definitions on the server with **Component Services**
- the remote server name on the client (this can also be checked with the OLE/COM Object Viewer)

When the connection has been established, you will be able to run your application on the client. Please remember that Broker ActiveX Control must be used as automation server. For information on how to use Broker ActiveX Control with Visual Basic see [Broker ActiveX Control as an Automation Server](#).

8

Publish and Subscribe with Broker ActiveX Control

- Writing Subscriber Applications 72
- Writing Publisher Applications 77

Broker ActiveX Control provides five Broker functions to enable publishing and subscription. Publish and subscribe enables an application to send a message (publication) to multiple receivers (subscribers).

This functionality is supported by the native COM interface as well as by the Transaction Object Repository interface (TOR file).

Some examples of publish and subscribe for the native interface are given below.

Writing Subscriber Applications

A subscriber receives the publications that are sent by the publisher. Subscribers will only receive publications that are sent *after* they have subscribed to a topic. Similarly, publishers can only send a publication if at least one subscriber has already subscribed to a topic.

To learn more about a particular topic, see *Writing Applications: Publish and Subscribe* in the ACI Programming documentation.

The methods, functions, properties and steps required to operate as subscriber are described here.

Methods

Method	Description
InvokeBrokerFunction	Invoke the Broker function call.
GetReceiveData	Return the most recently received publication as string.

Functions

These will be set in the `function` property. For all function calls, the `UserID`, `Password` (if security Broker), `Token` and `Topic` properties must be set.

Function	Option
Logon	
Subscribe	Option = None
Receive Publication	Option = None, Publication ID = NEW
Control Publication	Option = Commit
Unsubscribe	Option = None
Logoff	

Properties

Property	Description
APIVersion	Must be set to 8 or above
Function	See the Functions table.
Option	Needed to receive and control publications.
UserID	Your user ID.
Password	Your password.
Wait	Set an adequate amount of time to wait for a publication. The length of time depends on your application and can be set to Yes to wait until a publication has been received.
UOWstatus	Broker returns the current status of the publication.
ReceiveBufferLength	Set to the maximum possible publication length.
Token	Additional caller identifier. The combination of the user ID and the token must be unique.
Topic	The topic of the publication that is to be received. Use a topic that has been registered with the EntireX Broker. Ask your broker administrator to get a valid topic.
Publication ID	Always NEW for the first call to receive a publication. For subsequent messages, reuse the received publication ID. See the step Check UOW status to find out whether it is a multi-message publication.



Important: Please check the Error Status regularly; at least after every `InvokeBrokerFunction`.

▶ To operate as subscriber

- 1 Set the `APIVersion` property to 8, the functionality Publish and Subscribe is only available with API version 8 and above.
- 2 Set the `BrokerID` property for your EntireX Broker.
- 3 Set the `UserID`, `Password` (if required), `Token` and `Topic` properties.
- 4 Set the `Option` property to 0 (None).
- 5 Set the `Function` property to 9 (Logon). You must be logged on to use the publish-and-subscribe functionality.
- 6 Call the method `InvokeBrokerFunction` to perform the logon function. The application has now been logged on to the EntireX Broker.
- 7 After successful logon to the Broker, set the `function` property to 19 (subscribe).
- 8 Call the method `InvokeBrokerFunction` to subscribe. The application has now been subscribed as a non-durable subscriber for this topic. If you want to be a durable subscriber, set the `Option` property to **Durable** when calling the method `InvokeBrokerFunction`. To learn more about

the difference between durable and non-durable subscribers, see *Concepts of Persistent Messaging* in the general administration documentation.

- 9 Set the `Wait` property to the required value, for example 60s (s = seconds).
- 10 Set the `Option` property to 0 (None).
- 11 Set the `PublicationID` property to NEW.
- 12 Set the `Function` property to 18 (Receive Publication).
- 13 Set the `ReceiveBufferLength` property to your maximum expected publication length (can be up to 2048).
- 14 Call the method `InvokeBrokerFunction` to receive publications. The application will now wait to receive a publication. With the current settings the application would receive a publication within 60 seconds or time out after 60 seconds. If the publication is larger than 2048 characters, Broker ActiveX Control will return an error. Assuming that an application has received a publication, that publication now has a publication ID, assigned to it by the EntireX Broker.
- 15 Get the received data with the method `GetReceiveData`.
- 16 The current status of the publication is stored in the `UOWstatus` property. Check this `UOWstatus` now. A `UOWstatus` of 12 (Received Only), means that the received publication has only one message. A `UOWstatus` of 9 (Received First) means that you have received the first message of a multi-message publication. In this case you should request the other messages of this publication, until a `UOWstatus` of 11 (Received Last) is returned. See *Concepts of Persistent Messaging* in the general administration documentation for more information. To inform the EntireX Broker that the subscriber has received and retrieved the publication the subscriber must commit this.
- 17 Do not change the `PublicationID` property. This is required to refer to the received publication.
- 18 Set the `Option` property to 10 (Commit).
- 19 Set the `Function` property to 21 (**Control Publication**).
- 20 Call the method `InvokeBrokerFunction` to control the publication.
- 21 Get the `UOWstatus` property and check the status. The value of the `UOWstatus` should now be 5 (Processed). Your application may now run in a loop between steps 9 and 21 to receive several publications.
- 22 Set the `Option` property to 0 (None).
- 23 Set the `Function` property to 20 (Unsubscribe).
- 24 Call the method `InvokeBrokerFunction` to unsubscribe. The application has now been unsubscribed from the topic.
- 25 Set the `Function` property to 10 (Log off).
- 26 Call the method `InvokeBrokerFunction` to log off. The application has now been logged off from the EntireX Broker.

C# Example with a simple Subscriber who has Received only one Publication

```
using System;
// add the "EntireX Broker ActiveX Control" in COM references
using BrokerLib;

namespace Pubsub
{
    class Class1
    {
        static BrokerClass ebx;
        // EntireX Broker ACI definitions.
        const int function_logon = 9;
        const int function_logoff = 10;
        const int function_subscribe = 19;
        const int function_unsubscribe = 20;
        const int function_receive_publication = 18;
        const int function_control_publication = 21;
        const int option_none = 0;
        const int option_commit = 10;
        const int uowstatus_receive_only = 12;
        const int uowstatus_receive_last = 11;

        // procedure to invoke an entirex broker function call.
        static bool invokeEBX(short function, short option)
        {
            bool rc = true;
            ebx.Option = option;
            ebx.Function = function;
            ebx.InvokeBrokerFunction();
            // check the error status after the broker call.
            if (ebx.ErrorCode != "00000000")
            {
                Console.WriteLine(ebx.ErrorMessage);
                rc = false;
            }
            return rc;
        }

        [STAThread]
        static void Main(string[] args)
        {
            bool receive_error = false;
            bool subscribe_error = false;
            ebx = new BrokerClass();

            ebx.APIVersion = 8;
            ebx.BrokerID = "localhost";
            ebx.UserID = "EBXUSER";
            ebx.Token = "EBXTOKEN";
            ebx.Topic = "NYSE";
        }
    }
}
```

```
Console.WriteLine("Log on");
if (!invokeEBX(function_logon, option_none))
    return; // logon failed

Console.WriteLine("Subscribe");
if (!invokeEBX(function_subscribe, option_none))
    subscribe_error = true; // subscribe failed

if (!subscribe_error)
{
    ebx.PublicationID = "NEW";
    ebx.ReceiveBufferLength = 2048;
    ebx.Wait = "60s";
    // loop until all messages of the publication have been received.
    do
    {
        Console.WriteLine("Receive Publication");
        if (!invokeEBX(function_receive_publication, option_none))
        {
            receive_error = true; // receive failed
            break; // cancel the while loop
        }
        else
        {
            // work with the received publication.
            Console.WriteLine(ebx.GetReceiveData());
        }
    } while ((ebx.UOWStatus != uowstatus_receive_only) &&
        (ebx.UOWStatus != uowstatus_receive_last));

    if (!receive_error)
    {
        Console.WriteLine("Control Publication");
        invokeEBX(function_control_publication, option_commit);
        // the publication status should be 5 (= processed)
        Console.WriteLine("Publication status = " + ebx.UOWStatus);
    }
    Console.WriteLine("Unsubscribe");
    invokeEBX(function_unsubscribe, option_none);
}
Console.WriteLine("Log off");
invokeEBX(function_logoff, option_none);
}
}
```

Writing Publisher Applications

The publisher sends publications to subscribers. Publications will fail if there is no subscriber for this topic. See *Writing Applications: Publish and Subscribe* in the ACI Programming documentation for a list of the valid topics.

The methods, functions, properties and steps required to operate as Publisher are described below.

Methods

Method	Description
InvokeBrokerFunction	Invoke the broker function call.
SetSendData or SetSendDataLong	Set the publication to be sent.

Functions

These will be set in the `Function` property. For all function calls, the `UserID`, `Password` (if secure Broker), `Token` and `Topic` properties must be set.

Function	Option
Logon	
Send Publication	Option = Sync, Publication ID = NEW.
Control Publication	Option = Commit. A publication can also be committed with function=send_publication option=commit.
Logoff	

Properties

Property	Description
APIVersion	Must be set to 8 or above.
Function	See the function table.
Option	Needed to send and control publication.
UserID	Your user ID.
Password	Your password.
Wait	Must be set to NO.
UOWstatus	Broker returns the current status of the publication.
Token	Additional identifier of the caller. The combination of the user ID and the token must be unique.

Property	Description
Topic	The topic of the publication that is to be received. Use a topic that has been registered with the EntireX Broker. Ask your Broker administrator to get a valid topic.
PublicationID	Always NEW for the first call to send a publication. If you want to send a multi-message publication, reuse the received publication ID to send the other messages.



Important: Please check the Error Status regularly; at least after every `InvokeBrokerFunction`.

▶ **To operate as publisher**

- 1 Set the `APIVersion` property to 8, the publish-and-subscribe functionality is only available with API version 8 or above.
- 2 Set the `BrokerID` property for your EntireX Broker.
- 3 Set the `UserID`, `Password` (if required), `Token` and `Topic` properties.
- 4 Set the `Option` property to 0 (None).
- 5 Set the `Function` property to 9 (Logon). You must log on to use the publish-and-subscribe functionality.
- 6 Call the method `InvokeBrokerFunction` to perform the Logon function.

The application has now been logged on to the EntireX Broker.

- 7 Set the `Option` property to 10 (Commit).
- 8 Set the `Function` property to 17 (Send Publication).
- 9 Set the `Wait` property to NO.
- 10 Set the `PublicationID` property to NEW.
- 11 Call the method `SetSendData` or `SetSendDataLong` to set the publication data.
- 12 Call the method `InvokeBrokerFunction` to send the publication.
- 13 Get the `UOWstatus` property and check this. It should be 2 (Accepted).

A publication has now been sent. Please note that the publication will fail if there are no subscribers to this topic. If your publication has more than one message, the steps beginning with Set the `Option` property to 10 (Commit) will change. See *Concepts of Persistent Messaging* in the general administration documentation.

- 14 Set the `Option` property to 0 (None).
- 15 Set the `Function` property to 10 (Logoff).
- 16 Call the method `InvokeBrokerFunction` to log off.

The application has now been logged off from the EntireX Broker.

C# Example with a simple Publisher who Sends only one (single-message) Publication

```

using System;
// add the "EntireX Broker ActiveX Control" in COM references
using BrokerLib;

namespace Pubsub
{
    class Class1
    {
        static BrokerClass ebx;
        // EntireX Broker ACI definitions
        const int function_logon = 9;
        const int function_logoff = 10;
        const int function_send_publication = 17;
        const int function_control_publication = 21;
        const int option_none = 0;
        const int option_commit = 10;

        // procedure to invoke an entirex broker function call
        static bool invokeEBX(short function, short option)
        {
            bool rc = true;
            ebx.Option = option;
            ebx.Function = function;
            ebx.InvokeBrokerFunction();
            if (ebx.ErrorCode != "00000000")
            {
                Console.WriteLine(ebx.ErrorMessage);
                rc = false;
            }
            return rc;
        }

        [STAThread]
        static void Main(string[] args)
        {
            ebx = new BrokerClass();
            String s = "A small c# publisher example with EntireX Broker ActiveX
Control.";

            ebx.APIVersion = 8;
            ebx.BrokerID = "localhost";
            ebx.UserID = "EBXUSER";
            ebx.Token = "EBXTOKEN";
            ebx.Topic = "NYSE";

            Console.WriteLine("Log on");
            if (!invokeEBX(function_logon, option_none))
                return; // logon failed
        }
    }
}

```

```
    ebx.Wait = "NO";           // set to NO because we cannot receive data
    ebx.PublicationID = "NEW";
    ebx.SetSendDataLong(s, s.Length); // set the sent data

    Console.WriteLine("Send Publication");
    invokeEBX(function_send_publication, option_commit);
    // Check the status of the UOW. It should be 2 (= Accepted).
    Console.WriteLine("Publication status = " + ebx.UOWStatus);

    Console.WriteLine("Log off");
    invokeEBX(function_logoff, option_none);
}
}
```

9 Reference - Broker ActiveX Control

- Methods of Broker ActiveX Control 82
- Properties of Broker ActiveX Control 83

Methods of Broker ActiveX Control

This section describes the methods of Broker ActiveX Control.

Broker ACI

The following methods are useful for writing applications using the native interface.

Method	Description
BSTR GetReceiveData()	Return the received data inner string
BSTR GetErrorText()	Return the last received error message.
BOOL SetSendDataLong(String, Long) or BOOL SetSendData (String, Short)	Copy user's data buffer into the send buffer.
BOOL InvokeBrokerFunction()	Invoke the broker function call. Set the properties Function and Option.

Transaction Objects

Method	Description
Bool DefineTOMethods(String)	Starts the TO editor. If you specify a valid TOR name, this TO is then loaded into the editor. If a valid TOR name is not specified, the currently loaded TO will be displayed or an empty editor will be started.
Bool LoadTransObject(String)	Loads and initializes a transaction object. You must specify a valid TOR file name; otherwise FALSE will be returned.
Object CreateTransObject(String)	Loads and initializes a transaction object. You must specify a valid TOR file name. An object reference will be returned, which can be used to call the methods defined in the TO. If loading fails, a null reference will be returned.
Object CreateTransObjectSA(String)	This method uses the safe array implementation for arrays instead of the collection implementation. If you experience problems accessing arrays with an automation controller, try using this method to instantiate a TOR object.

Properties of Broker ActiveX Control

Most properties of Broker ActiveX Control correspond to the Broker ACI fields. The properties must be set to the appropriate values before using any function.

If transaction object repository (TOR) files are used, it will not be necessary to set all the properties. See section [Transaction Objects in Broker ActiveX Control](#). The properties can also be supplied by means of the property pages (see [Using the Property Pages](#) in section *Writing Applications - Broker ActiveX Control*).

Property Name	Broker ACI Field	Format	Length	API Version	Description
AdapterError	not used	String	8	2	
AdCount	not used	Long		2	
APIVersion	API-VERSION	Short		2	<p>Possible values: 1, 2, 3, 4, 5, 6, 7, 8, 9.</p> <p>The default is 2. This value can be changed dynamically by setting the property. If the current value of the <code>Function</code> or <code>Option</code> property requires a minimal API version, the value of <code>APIVersion</code> will be adjusted automatically.</p>
BrokerID	BROKER-ID	String	32	1	Target Broker ID. See <i>Using the Broker ID in Applications</i> in the ACI Programming documentation and details on TCP/IP in <i>Transport Methods</i> under <i>Writing Applications: Client and Server Publish and Subscribe</i> in the ACI Programming documentation.
BrokerSecurity	KERNELSECURITY	String	1	7	
ClientUserid	CLIENT-UID	String	32	2	The partner's user ID.

Property Name	Broker ACI Field	Format	Length	API Version	Description						
CommitTime	COMMITTIME	String	17	7	Readonly property. Time when UOW was committed. Format: YYYYMMDDHHMMSSms ms = milliseconds in Possible Values field.						
CompressLevel	COMPRESSLEVEL	String	1	7	Compression level. Possible values: N/Y/0-9. The first character of the string will be used as the compression value. If you type YES, the character Y will be used and ES will be cut off. Example: Broker1.CompressLevel = "6". See also <i>Data Compression</i> under <i>Writing Applications: Client and Server Publish and Subscribe</i> in the ACI Programming documentation.						
ConvID	CONV - ID	String	16	1	Conversation ID, see <i>Managing Conversation Contexts</i> under <i>Writing Applications: Client and Server</i> in the EntireX Broker ACI Programming documentation.						
ConvStatus	CONV - STAT	Short		2	Contains the status of the conversation when the RECEIVE function is complete. See <i>Managing Conversation Contexts</i> under <i>Writing Applications: Client and Server</i> in the EntireX Broker ACI Programming documentation. Possible values: <table border="1" data-bbox="1096 1753 1421 1890"> <tr> <td>1</td> <td>NEW</td> </tr> <tr> <td>2</td> <td>OLD</td> </tr> <tr> <td>3</td> <td>NONE</td> </tr> </table>	1	NEW	2	OLD	3	NONE
1	NEW										
2	OLD										
3	NONE										

Property Name	Broker ACI Field	Format	Length	API Version	Description																																
EncryptionLevel	ENCRYPTION - LEVEL	Short		6	Possible values: 0, 1, 2. See <i>Encryption</i> under <i>Writing Applications using EntireX Security</i> in the ACI Programming documentation.																																
Environment	ENVIRONMENT	String	32	1																																	
ErrorCode	ERROR - CODE	String	8	1	Broker error code, see <i>Error Handling</i> under <i>Writing Applications: Client and Server Publish and Subscribe</i> in the ACI Programming documentation.																																
ErrorMsg	not used	String	40	1	Contains the error message to the corresponding error code.																																
ForceLogon	FORCE - LOGON	Boolean		6	Possible values: Y, N.																																
Function	FUNCTION Possible values: <table border="1" data-bbox="537 999 906 1730"> <tr><td>1</td><td>SEND</td></tr> <tr><td>2</td><td>RECEIVE</td></tr> <tr><td>4</td><td>UNDO</td></tr> <tr><td>5</td><td>EOC</td></tr> <tr><td>6</td><td>REGISTER</td></tr> <tr><td>7</td><td>DEREGISTER</td></tr> <tr><td>8</td><td>VERSION</td></tr> <tr><td>9</td><td>LOGON</td></tr> <tr><td>10</td><td>LOGOFF</td></tr> <tr><td>13</td><td>SYNCPOINT</td></tr> <tr><td>14</td><td>KERNELVERS</td></tr> <tr><td>17</td><td>SEND_PUBLICATION</td></tr> <tr><td>18</td><td>RECEIVE_PUBLICATION</td></tr> <tr><td>19</td><td>SUBSCRIBE</td></tr> <tr><td>20</td><td>UNSUBSCRIBE</td></tr> <tr><td>21</td><td>CONTROL_PUBLICATION</td></tr> </table>	1	SEND	2	RECEIVE	4	UNDO	5	EOC	6	REGISTER	7	DEREGISTER	8	VERSION	9	LOGON	10	LOGOFF	13	SYNCPOINT	14	KERNELVERS	17	SEND_PUBLICATION	18	RECEIVE_PUBLICATION	19	SUBSCRIBE	20	UNSUBSCRIBE	21	CONTROL_PUBLICATION	Short		1	The functions to be performed by Broker.
1	SEND																																				
2	RECEIVE																																				
4	UNDO																																				
5	EOC																																				
6	REGISTER																																				
7	DEREGISTER																																				
8	VERSION																																				
9	LOGON																																				
10	LOGOFF																																				
13	SYNCPOINT																																				
14	KERNELVERS																																				
17	SEND_PUBLICATION																																				
18	RECEIVE_PUBLICATION																																				
19	SUBSCRIBE																																				
20	UNSUBSCRIBE																																				
21	CONTROL_PUBLICATION																																				
LocaleString	LOCALE - STRING	String	40	4	For sending locale strings to the broker (see Using Internationalization in																																

Property Name	Broker ACI Field	Format	Length	API Version	Description																																												
					<i>Writing Applications - Broker ActiveX Control).</i>																																												
MessageId	not used	String	32	2																																													
MessageType	not used	String	32	2																																													
NewPassword	NEWPASSWORD	String	32	2																																													
Option	OPTION Possible values: <table border="1"> <tr><td>0</td><td>NULL</td></tr> <tr><td>1</td><td>MSG</td></tr> <tr><td>2</td><td>HOLD</td></tr> <tr><td>3</td><td>IMMED</td></tr> <tr><td>4</td><td>QUIESCE</td></tr> <tr><td>5</td><td>EOC</td></tr> <tr><td>6</td><td>CANCEL</td></tr> <tr><td>7</td><td>LAST</td></tr> <tr><td>8</td><td>NEXT</td></tr> <tr><td>9</td><td>PREVIEW</td></tr> <tr><td>10</td><td>COMMIT</td></tr> <tr><td>11</td><td>BACKOUT</td></tr> <tr><td>12</td><td>SYNC</td></tr> <tr><td>13</td><td>ATTACH</td></tr> <tr><td>14</td><td>DELETE</td></tr> <tr><td>15</td><td>EOCANCEL</td></tr> <tr><td>16</td><td>QUERY</td></tr> <tr><td>17</td><td>SETUSTATUS</td></tr> <tr><td>18</td><td>ANY</td></tr> <tr><td>19</td><td>no longer used</td></tr> <tr><td>20</td><td>DURABLE</td></tr> <tr><td>21</td><td>CHECKSERVICE</td></tr> </table>	0	NULL	1	MSG	2	HOLD	3	IMMED	4	QUIESCE	5	EOC	6	CANCEL	7	LAST	8	NEXT	9	PREVIEW	10	COMMIT	11	BACKOUT	12	SYNC	13	ATTACH	14	DELETE	15	EOCANCEL	16	QUERY	17	SETUSTATUS	18	ANY	19	no longer used	20	DURABLE	21	CHECKSERVICE	Short		1	
0	NULL																																																
1	MSG																																																
2	HOLD																																																
3	IMMED																																																
4	QUIESCE																																																
5	EOC																																																
6	CANCEL																																																
7	LAST																																																
8	NEXT																																																
9	PREVIEW																																																
10	COMMIT																																																
11	BACKOUT																																																
12	SYNC																																																
13	ATTACH																																																
14	DELETE																																																
15	EOCANCEL																																																
16	QUERY																																																
17	SETUSTATUS																																																
18	ANY																																																
19	no longer used																																																
20	DURABLE																																																
21	CHECKSERVICE																																																
Password	PASSWORD	String	32	1																																													
ReceiveBufferLength	RECEIVE - LENGTH	Long		3	Length of the receive buffer.																																												
ReceiveBufferSize	RECEIVE - LENGTH	Short		1	This is an old property. Can be used instead of ReceiveBufferLength - for buffers with less than 32 KB only.																																												

Property Name	Broker ACI Field	Format	Length	API Version	Description																										
ReturnDataLength	RETURN-LENGTH	Long		3	Length of returned data.																										
ReturnLength	RETURN-LENGTH	Short		1	This is an old property. Can be used instead of ReturnDataLength - for buffers with less than 32 KB only.																										
SecurityToken	SECURITY-TOKEN	String	32	2	This is handled automatically, but can be filled in by the user if required.																										
SendBufferSize		Short		1	No longer used.																										
ServerClass	SERVER-CLASS	String	32	1	These three Broker parameters form the target service.																										
ServerName	SERVER-NAME	String	32	1																											
Service	SERVICE	String	32	1																											
Store	STORE	Short		2	Possible values: <table border="1" data-bbox="1198 884 1421 1014"> <tr> <td>0</td> <td>NULL</td> </tr> <tr> <td>1</td> <td>OFF</td> </tr> <tr> <td>2</td> <td>BROKER</td> </tr> </table>	0	NULL	1	OFF	2	BROKER																				
0	NULL																														
1	OFF																														
2	BROKER																														
Token	TOKEN	String	32	1																											
UOWID	UOWID	String	16	3																											
UOWStatus	UOWSTATUS	Short		3	Possible values: <table border="1" data-bbox="544 1230 820 1818"> <tr> <td>0</td> <td>NONE</td> </tr> <tr> <td>1</td> <td>RECEIVED</td> </tr> <tr> <td>2</td> <td>ACCEPTED</td> </tr> <tr> <td>3</td> <td>DELIVERED</td> </tr> <tr> <td>4</td> <td>BACKEDOUT</td> </tr> <tr> <td>5</td> <td>PROCESSED</td> </tr> <tr> <td>6</td> <td>CANCELLED</td> </tr> <tr> <td>7</td> <td>TIMEOUT</td> </tr> <tr> <td>8</td> <td>DISCARDED</td> </tr> <tr> <td>9</td> <td>FIRST</td> </tr> <tr> <td>10</td> <td>MIDDLE</td> </tr> <tr> <td>11</td> <td>LAST</td> </tr> <tr> <td>12</td> <td>ONLY</td> </tr> </table>	0	NONE	1	RECEIVED	2	ACCEPTED	3	DELIVERED	4	BACKEDOUT	5	PROCESSED	6	CANCELLED	7	TIMEOUT	8	DISCARDED	9	FIRST	10	MIDDLE	11	LAST	12	ONLY
0	NONE																														
1	RECEIVED																														
2	ACCEPTED																														
3	DELIVERED																														
4	BACKEDOUT																														
5	PROCESSED																														
6	CANCELLED																														
7	TIMEOUT																														
8	DISCARDED																														
9	FIRST																														
10	MIDDLE																														
11	LAST																														
12	ONLY																														
UOWStatusPersist	UOW-STATUS-PERSIST	Short		3																											

Property Name	Broker ACI Field	Format	Length	API Version	Description
UOWTime	UWTIME	String	8	3	
UserData	USER-DATA	String	16	2	This field is not converted by the Broker. If the field contains H'00', only the data up to the first H'00' will be sent.
UserID	USER-ID	String	32	1	User ID.
UserStatus	USTATUS	String	32	3	
Wait	WAIT	String	8	1	Possible values: Yes No <n>S - waiting n Seconds (max 99999) <n>M - waiting n Minutes (Max 99999) <n>H - waiting n Hours (max 99999). See <i>Blocked and Non-blocked Broker Calls under Writing Applications: Client and Server in the EntireX Broker ACI Programming documentation.</i>
Topic		String		8	Required for handling with publish and subscribe.
PublicationID		String	16	8	Required for handling with publish and subscribe.
UOWStatusLife		String	8	8	