Transaction Objects in Broker ActiveX Control

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.

This chapter covers the following topics:

- Advantages of Transaction Objects
- Calling the Transaction Object Editor
- Managing TOR Files
- Defining Methods
- Specifying Connection Information
- Defining Custom Data Types
- TOR Files in IDL Format
- TOR Files in XML Format
- Storing TOR Files in a Tamino Database

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.

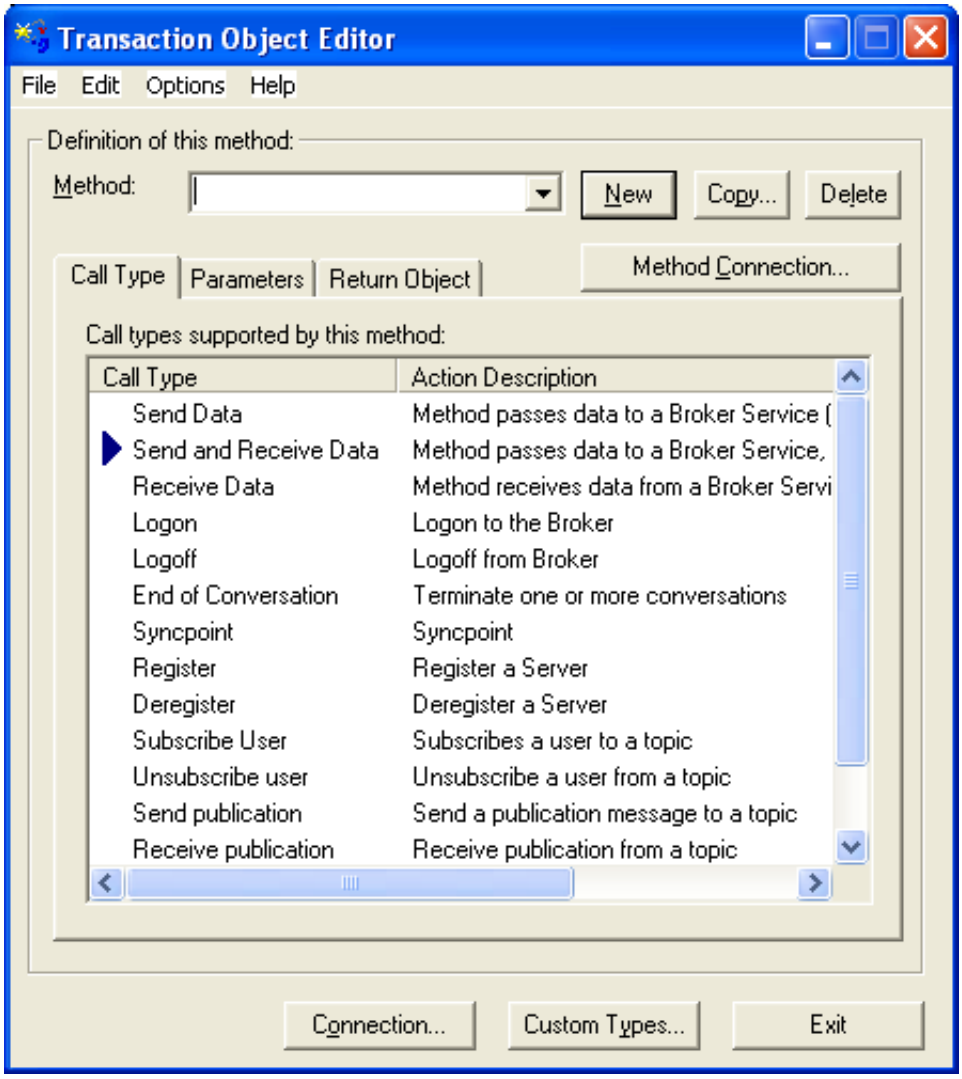
Calling the Transaction Object Editor

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.



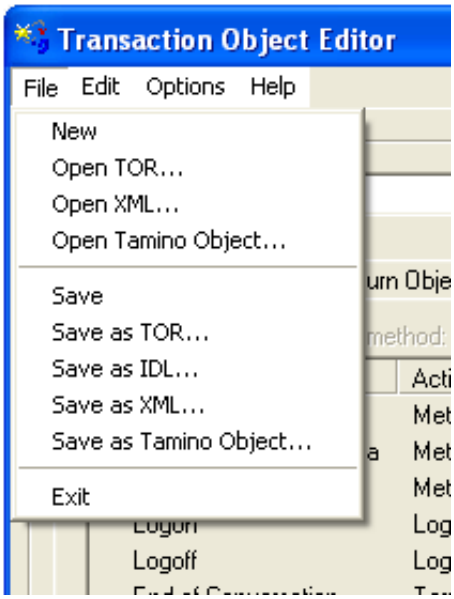
When a transaction object is loaded, the corresponding file name will be displayed in the title bar. If loading or saving fails, an error message will be displayed in the title bar.

Managing TOR Files

The following functions are available for managing TOR files.

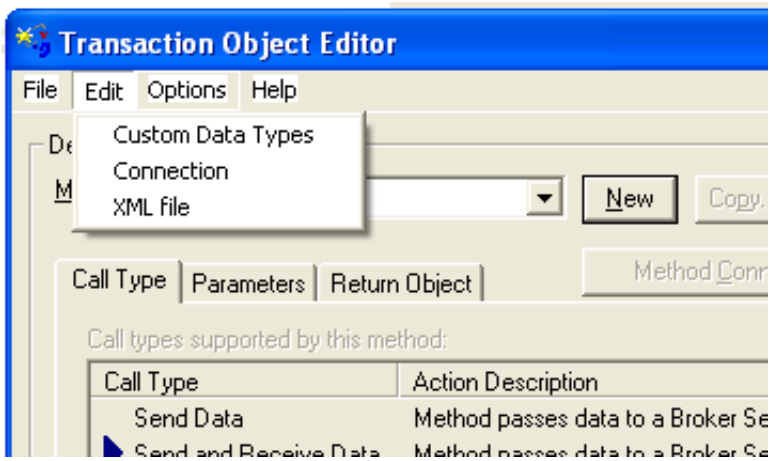
- File Menu
- Edit Menu
- Options Menu
- Help Menu

File Menu



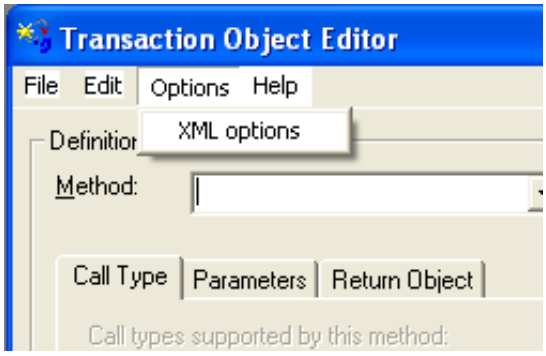
Menu Item	Description
New	Resets the TOR Editor.
Open TOR	Loads an existing TOR file. A standard **Open File** dialog will be displayed. This function is needed to modify an existing TOR file.
Open XML	Loads an existing XML file. A standard **Open File** dialog will be displayed. This function is needed 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.

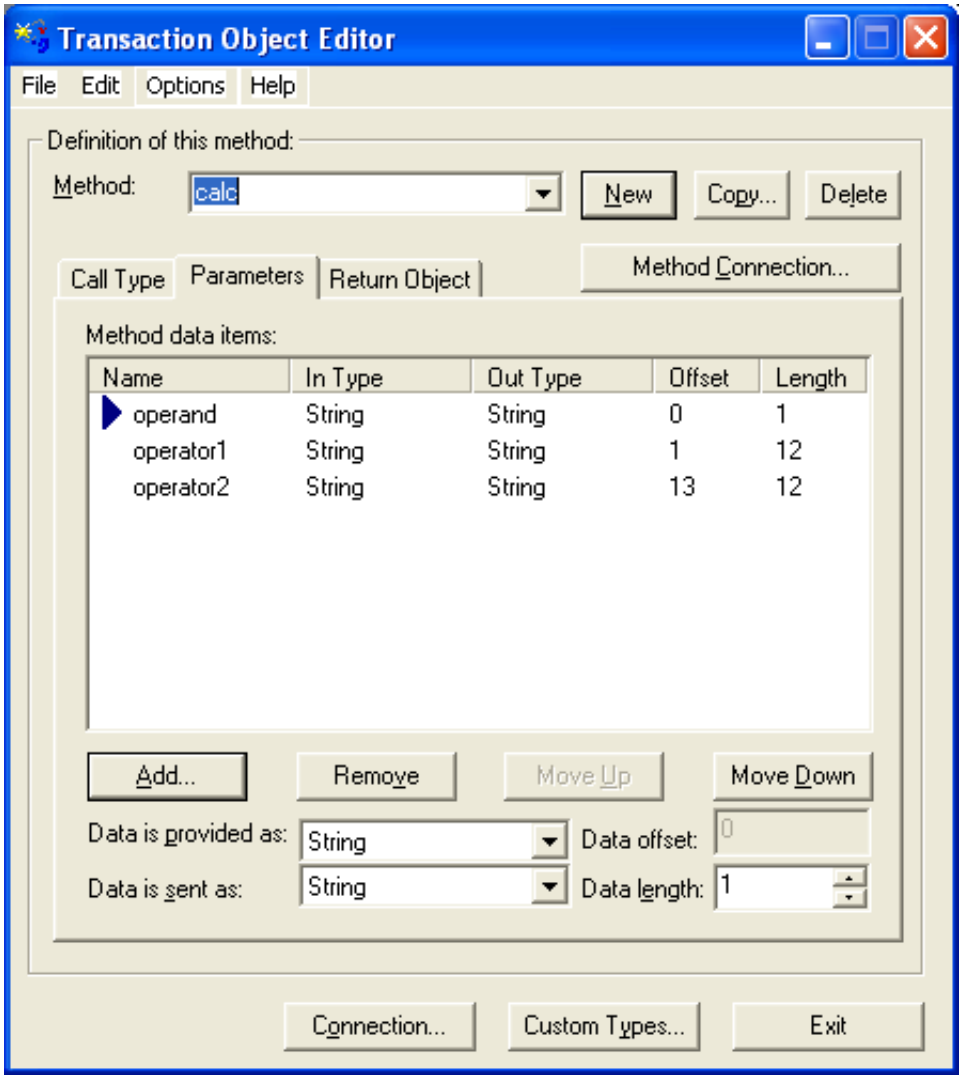
Call Type	Description
Send Data	Used to define a method that accepts parameters but does not return data from the service. This could be used to notify a back-end application of some event without waiting for a response.
Send and Receive Data	Used to define a method that accepts parameters and returns 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 `TO` 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 `TO` 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

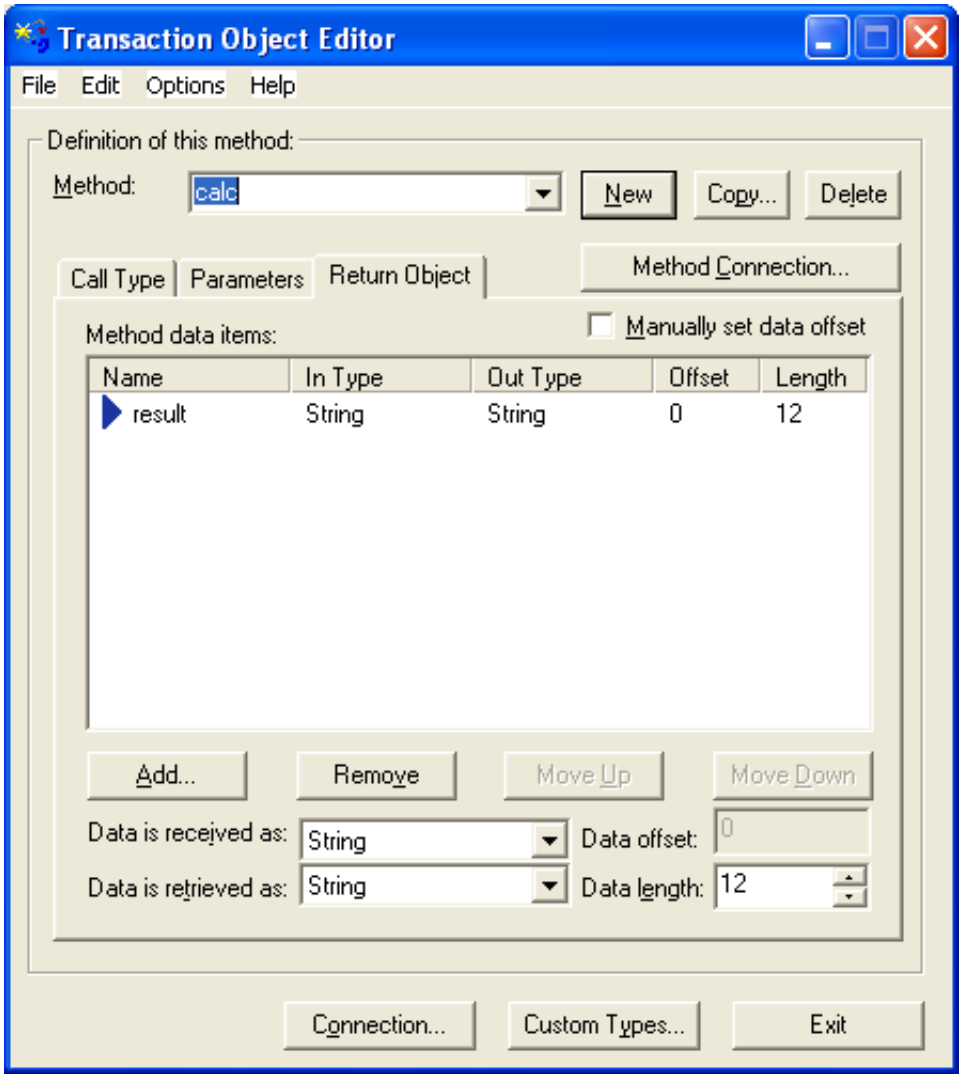
The scalar data types supported by the Broker ActiveX are a subset of the standard Automation VARIANT types and are listed below. In cases where the selected data type is of fixed length, the data length edit control is set to the appropriate length and grayed.

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.
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.

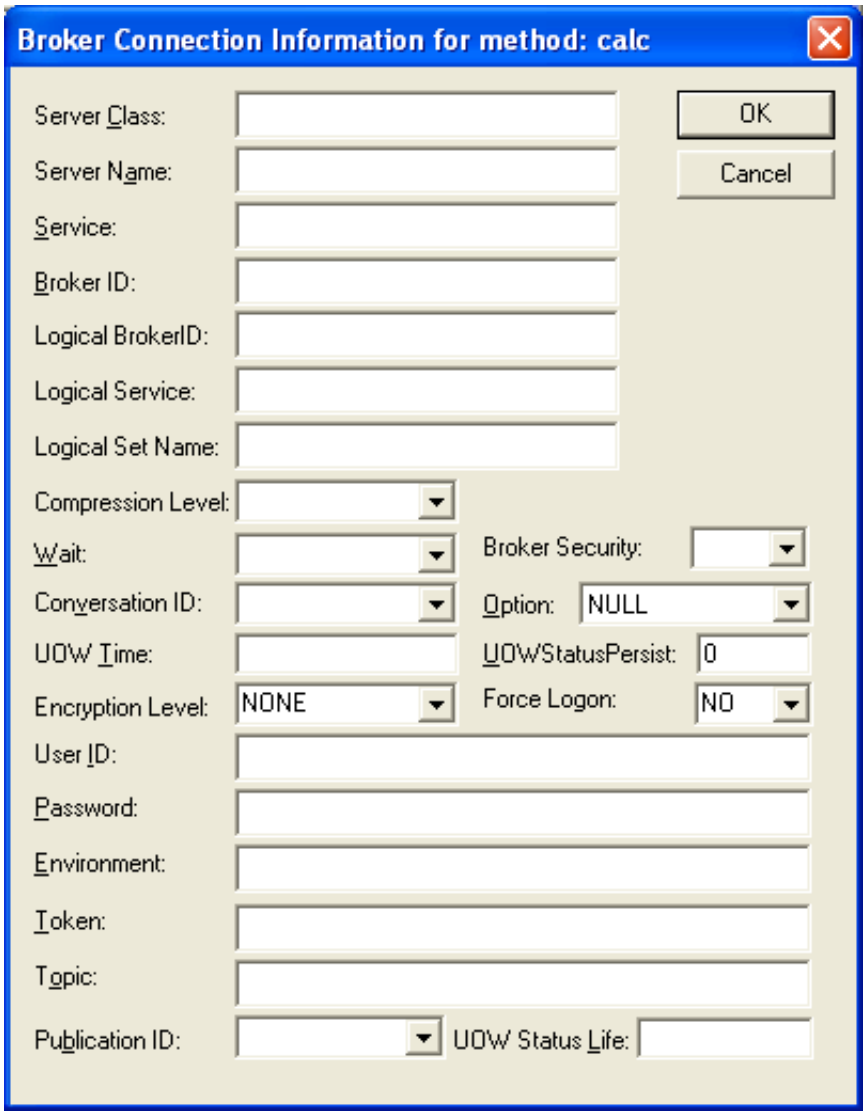
The **Manually set data offset** check box allows the transaction object designer to override automatic offset calculation and specify offsets manually. This feature is powerful, but also potentially dangerous, because no base type checking can be performed.

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 **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 `BrokerID` 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 *Data Compression in EntireX Broker*.
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: **Operation** — Wait Value (in seconds) Send — 0 Send and Receive — 30 (*) Receive — 59 (*) (*) if no value is specified in the **Connection** info.

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’

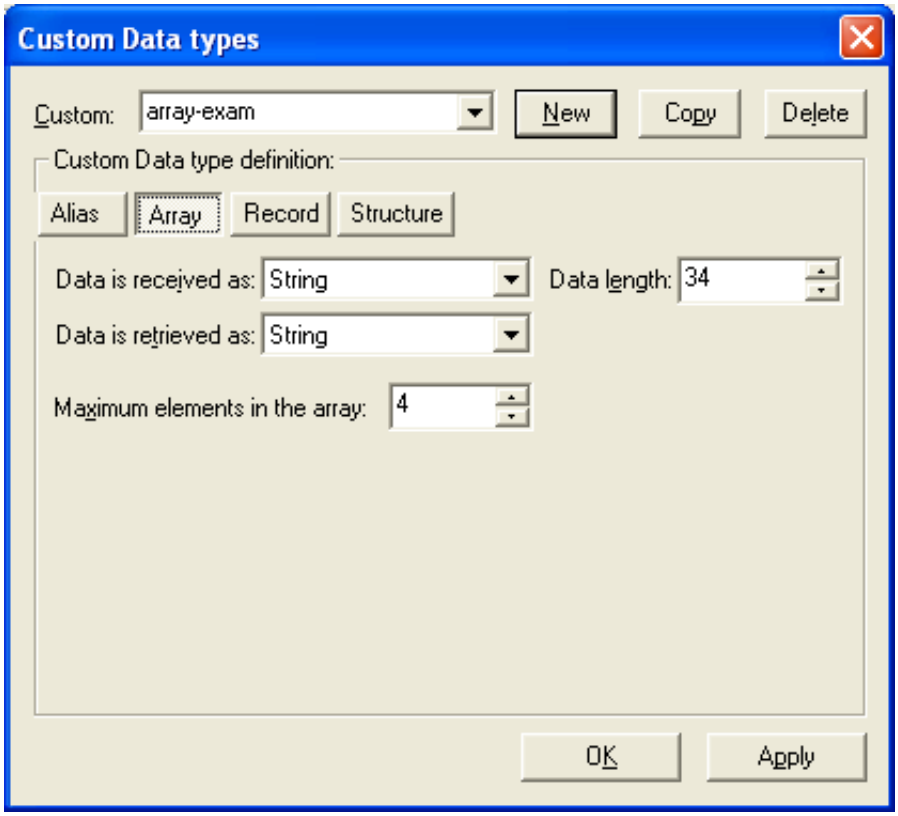
Any custom data type can be used in transaction objects return objects. Custom data types are not supported as method parameters.

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 ’



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.

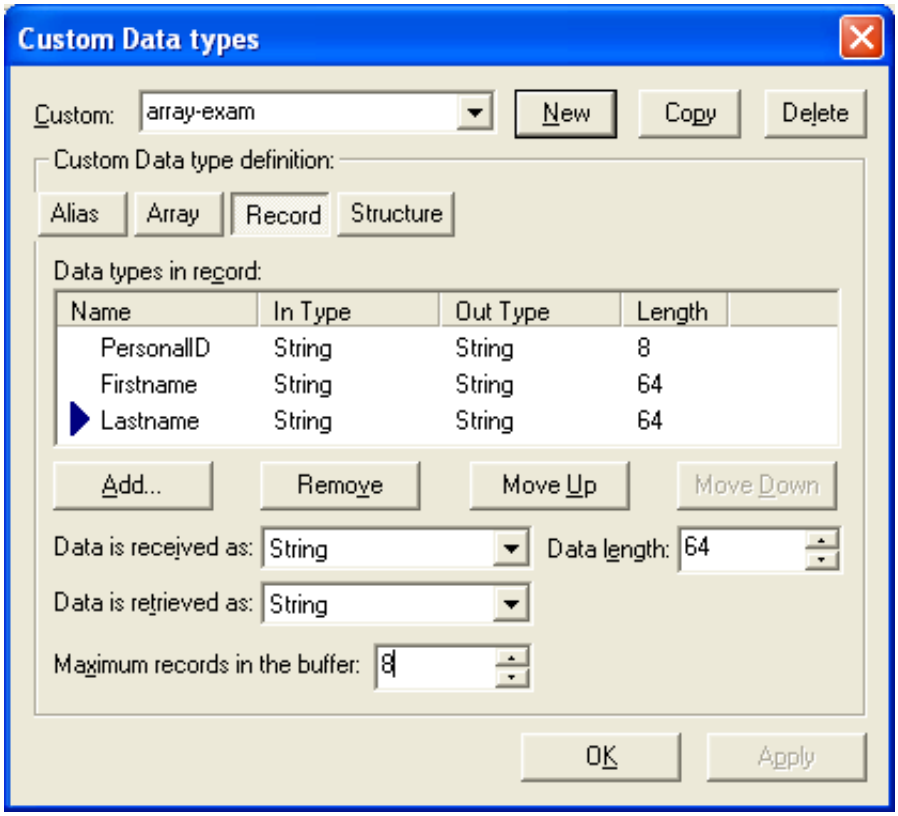
Array custom data types accept the same basic information as alias data types, with the addition of the number of elements in the array. Arrays allow elements of the specified base type to be accessed in a subscripted fashion.

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.



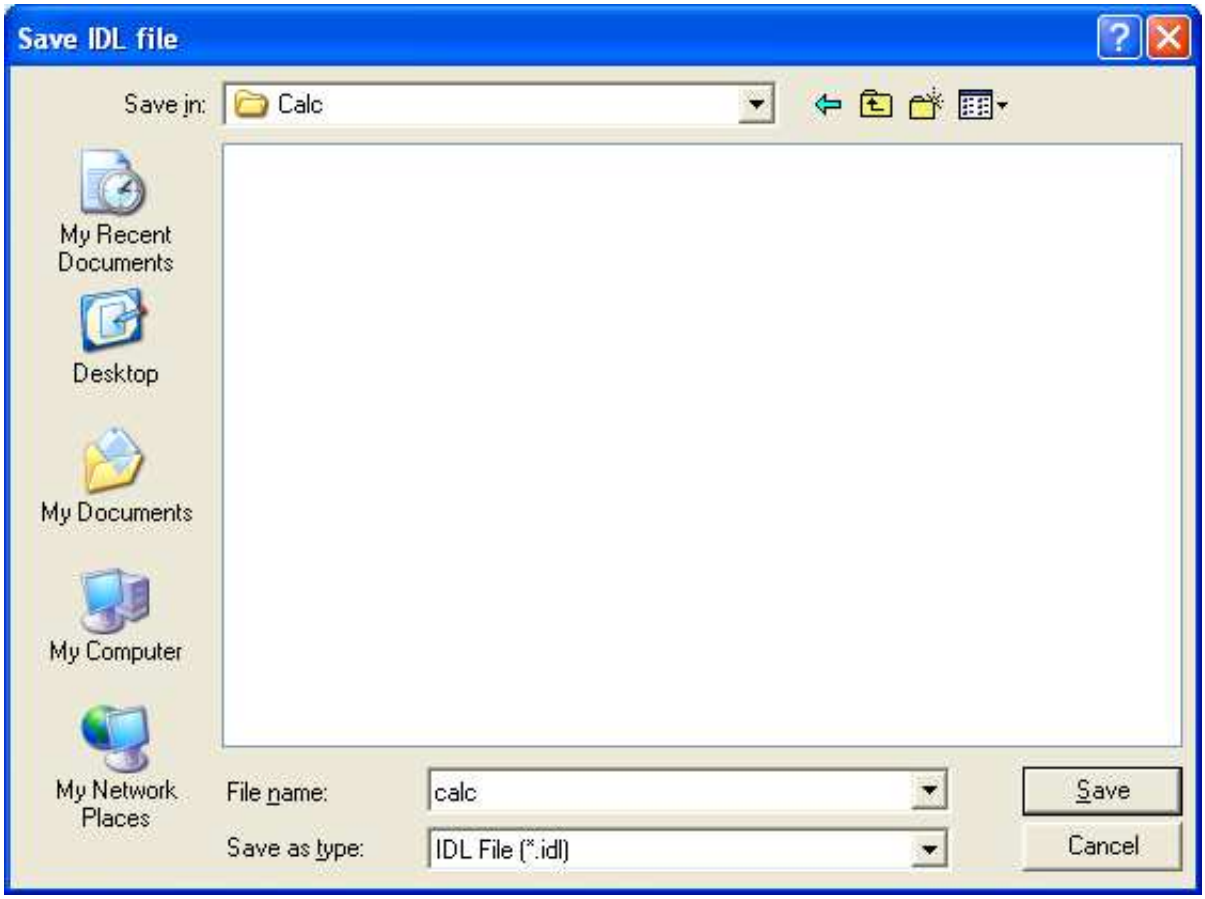
Custom Data Type ’Structure’

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
`<!ELEMENT EntireXTorFile (DefaultConnection? , Method*, CDT*)>`	The root must always be defined. It contains: ● 0-1 default connections ● 0-n methods ● 0-n CDT (= custom data types)
`<!ATTLIST EntireXTorFile Name CDATA #IMPLIED` `Version CDATA #IMPLIED>`	**Name** The name of the TOR file. **Version** The EntireX version with which the XML file was generated
`<!ELEMENT DefaultConnection EMPTY>`	The global connection information is stored here.
`<!ATTLIST DefaultConnection %Connection;>`	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
`<!ELEMENT Method (MethodConnection? , Parameter*)>`	Each method contains: ● 0-1 method connections ● 0-n parameter

`<!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>`
	A name and a call type must be defined for each method. The manual offset contains the manual offset switch of the return map.

`<!ELEMENT MethodConnection EMPTY>`	The connection information of each method is stored here.
`<!ATTLIST MethodConnection %Connection;>`	All parameters belonging to the method connection are stored as attributes. See the detailed description of the %Connection at the end of this table.
`<!ELEMENT Parameter (InFormat, OutFormat, Length?)>`	Each parameter contains: ● 1 in format ● 1 out format ● 0-1 length

`<!ATTLIST Parameter Name CDATA #IMPLIED`
`Direction (IN | OUT | INOUT) #IMPLIED`
`Offset CDATA #IMPLIED>`
	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

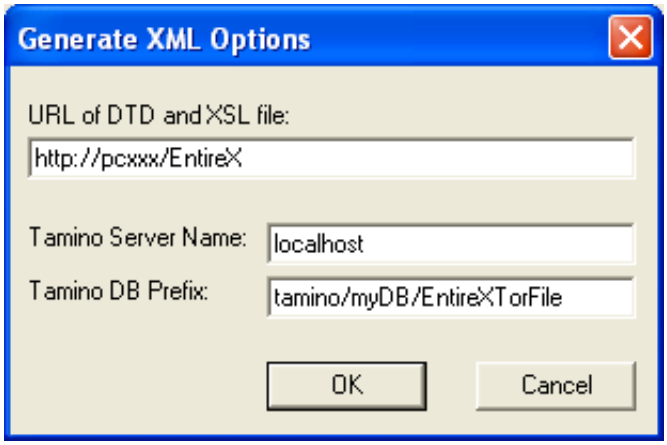
`<!ELEMENT CDT (Alias | Array | Record | Structure) >`	A custom data type (CDT) is an alias, an array, a record or a structure.

Entry in the DTD file	Explanation
`<!ATTLIST CDT Name ID #REQUIRED>`	The name of the CDT is required.
`<!ELEMENT Alias (InFormat, OutFormat, Length?)>`	An alias contains: • 1 in format • 1 out format • 0-1 length
`<!ELEMENT Array (InFormat, OutFormat, Length?)>`	An array contains: • 1 in format • 1 out format • 0-1 length
`<!ATTLIST Array NumberEle CDATA #IMPLIED>`	The numbers of elements for an array are stored here.
`<!ELEMENT Record (Parameter*)>`	The record contains: • 0-n parameter
`<!ATTLIST Record NumberEle CDATA #IMPLIED>`	The numbers of elements for a record are stored here.
`<!ELEMENT Structure (Parameter*) >`	The structure contains: • 0-n parameter
`<!ELEMENT InFormat (Scalar | UsedCDT)>`	An InFormat is a scalar value or a reference to a CDT.
`<!ELEMENT Scalar EMPTY>`	
`<!ATTLIST Scalar Format (I1 | I2 | I4 | F4 | F8 | Bool | String | Blob | Padding) #REQUIRED>`	A scalar must be in one of the listed formats.
`<!ELEMENT UsedCDT EMPTY>`	
`<!ATTLIST UsedCDT Target IDREF #REQUIRED>`	A UsedCDT must reference the name of a defined CDT.
`<!ELEMENT OutFormat (Scalar | UsedCDT)>`	An OutFormat is a scalar value or a reference to a CDT.
`<!ELEMENT Length EMPTY>`	
`<!ATTLIST Length Value CDATA #IMPLIED>`	A length must be defined for scalars with the values: string, BLOB and padding or UsedCDTs.

Entry in the DTD file	Explanation
<!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 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" >	All connection parameters are defined as attributes.

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

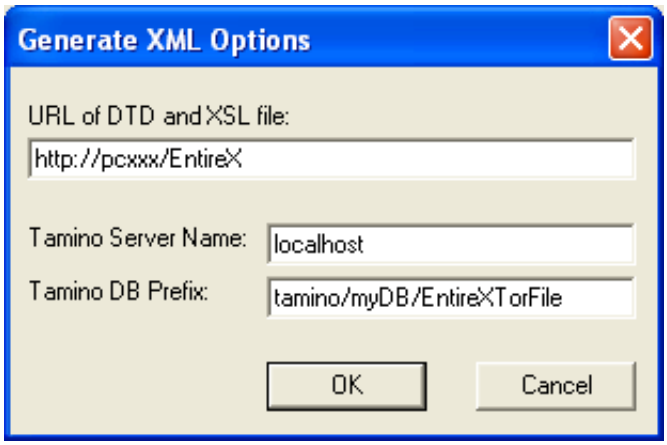
```
Set TransObject=BOCX.CreateTransObject ("...\\calc.xml")
```

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.

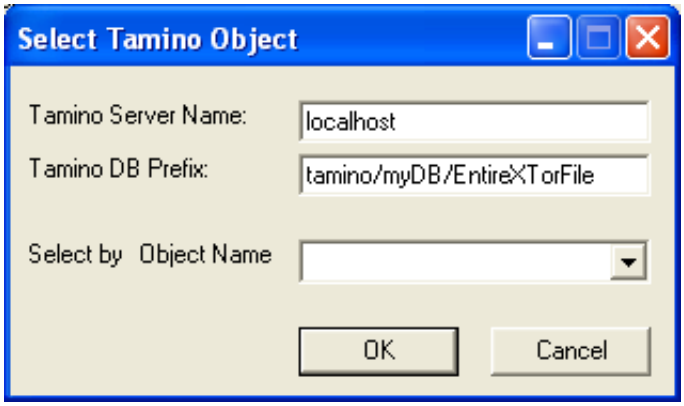
Creating a Tamino Database for the TOR Files

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.



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.