TCP Bridge Programmer's Guide

EsiObjects V4.0

ESI Technology Corporation

5 Commonwealth Road

Natick, MA. 01760

www.esitechnology.com

Table of Contents

Introduction

This guide is designed to assist the EsiObjects programmer in using the TCP Bridge to build object oriented application systems. It contains the following:

- An overview of the TCP Bridge.
- The concepts of the TCP Bridge.
- Description and instructions on how to use the basic bridge operations.
- How to perform lifespan operations.
- How to use the bridge with proxies.
- Description and how to use Bulk Transfer Objects in conjuction with the bridge.
- How to use event processing through the bridge.
- How to use some advanced features.
- How to access the model side functionality from Active Server Pages.
- Reference information on the bridge itself.

Document Convention

EsiObjects documentation uses the following typographical conventions:

For more information on this subject please refer to the BREAK Command section of this Guide.	Underlined text is used to highlight a reference to another section of this manual or another guide.
Property	In text, italicized words indicate defined terms that are usually used for the first time. Words are also italicized for emphasis.
CREATE	Words in bold and capitalized are EsiObjects commands or keywords.
`Set T%Test=I%Pat.Name`	This font is used for code examples.

TCP Bridge Overview

What is the TCPBridge?

The TCPBridge is an ActiveX® component. ActiveX controls are among the many types of components that use COM technologies to provide interoperability with other types of COM components and services. In EsiObjects, this COM component allows for interoperability between COM components such as Visual Basic (VB) forms and the EsiObjects server. This connection between the COM component and EsiObjects is done via TCP/IP.

The bridge is actually made up of several components that allow for efficient and specialized services to the server. These components are all described in this document.

Using the bridge, a COM component or service can connect to the EsiObjects server and have access to the basic services in EsiObjects: object creation and destruction, operations on methods, properties, and relationships, events and more. The bridge supports only one simultaneous connection to a server. Multiple bridges may be used to make multiple connections if needed.

The bridge, when connected to the server, occupies one M process that handles the communications between the bridge and EsiObjects. Thus the objects created within this process are valid only in the bridge that issued them, unless the object is created as a *shared* object (more on this later.)

Scope of Document

This document describes the basic usage of the bridge and gives examples using Visual Basic. Using this guide you will learn how to:

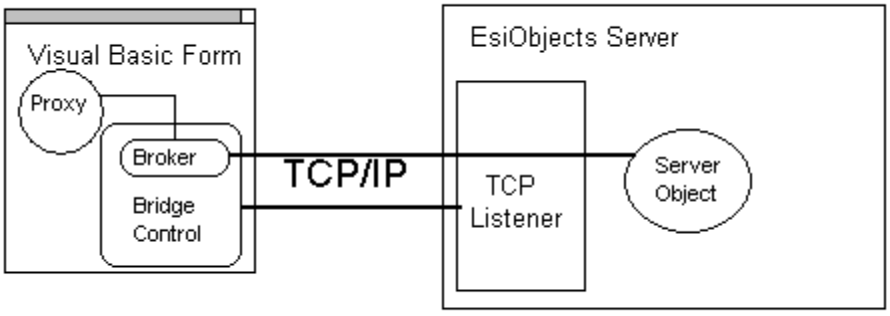
- Connect and disconnect a COM component (a Visual Basic application) to an EsiObjects server.
- Lookup, create and destroy objects.
- Use proxy objects on the client to invoke methods, set and get property values, set relationships on server objects.
- Use event processing.
- Use bulk data transfer objects that are supplied with the bridge to allow the transfer of bulk data across from the server for manipulation on the client, and then back to the server.
- Use the bridge to access objects on the server via an API instead of proxies (earlier versions of the bridge did not use proxies.)
- Access the EsiObjects server from Active Server Pages (ASP.)
- The document also includes a reference guide that details the services provided by the bridge and its components.

Concepts

Communications Diagram

The diagram below shows how the TCP Bridge relates to the EsiObjects server. The control is loaded onto a VB form. The control is connected to a TCP Listener process running on an EsiObjects server. Once connected, the VB programmer can access the Broker object within the control. The Broker is the object that routes requests from the client to the server. A VB Proxy object maps to a server object. Operations invoked on the proxy are sent, via the broker, to the servant object. The operation is executed on that object any result is returned back to the proxy.

Not shown in the diagram are the Bulk Transfer objects that the bridge provides for retrieving and sending collections of data to and from the server. These objects are described later in this guide.



The ActiveX control is loaded on a VB form and uses TCP/IP to connect to a TCP listener that is running on the EsiObjects server. Proxies are used to invoke operations on objects within the server.

The Bridge, the Broker, and Proxies

The Bridge

The Bridge object in the control provides a number of operations and properties for connecting the VB application to EsiObjects. These operations and properties, listed in the Reference section below, are used to connect to and disconnect from EsiObjects. This object is the root object for connecting to the server. It must be connected first before the bridge can be used with EsiObjects.

The Broker

The Broker object is the main object used to access EsiObjects functionality. It handles object requests (invoking methods, setting properties, etc.) from the client to the server.

Proxies

Proxies are generic COM objects that provide direct access to an EsiObjects object. Calls made on a proxy (for example, setting a property) are forwarded to the server for processing. The proxy provides a direct mapping between an EsiObjects object and a COM object. These proxies are actually created by the Broker (transparent to the programmer.) Thus all proxies share a common connection to the server via the broker. Invoking an operation on a proxy will cause the corresponding operation on the servant object to be executed.

Data Types

Most data is passed to and returned from EsiObjects as Variants. A handle to a server object is transported by its object id. In addition, the control provides a number of data types for bulk transfer objects that are used to provide an In/Out call dynamic.

What is EOSupport

Within the control there is a supplied library *EOSupport* that provides a number of COM objects to support bulk transfer operations, including large text objects, and streams. These objects are described in detail later in this guide.

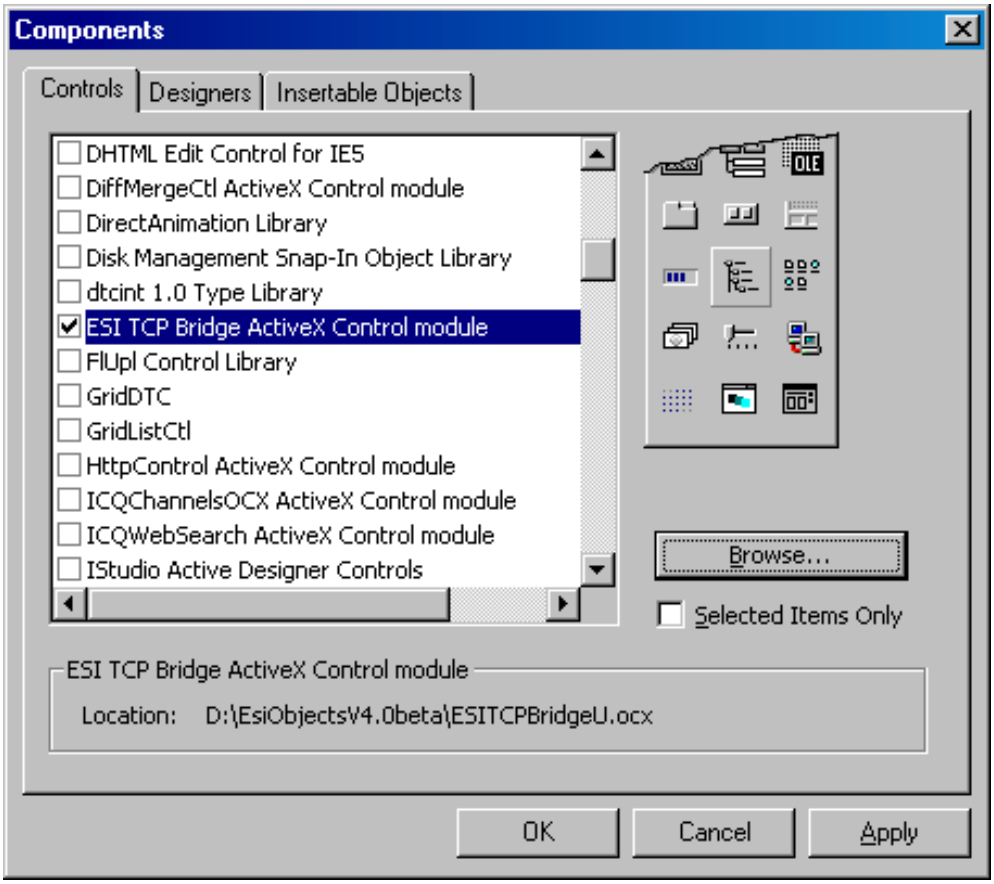
Basic Operations

Getting Started in VB

This guide describes the use of the TCP Bridge within a Visual Basic environment. Please refer to your product documentation if you are using another product that uses ActiveX controls.

Adding the TCPBridge to the Components Tab

Once the control has been installed from the EsiObjects installation kit, and registered, you should add the control to the components list within VB. This list contains all the controls available to you for your use in VB.



The Components Dialog

To add the control to the component list, perform the following steps:

1. From the **Project** menu, select the **Components** option.
2. If your control was properly installed and registered, you should see the control listed as “ESI TCP Bridge Active X Control Module”. Check this item (as shown above).
3. Select OK

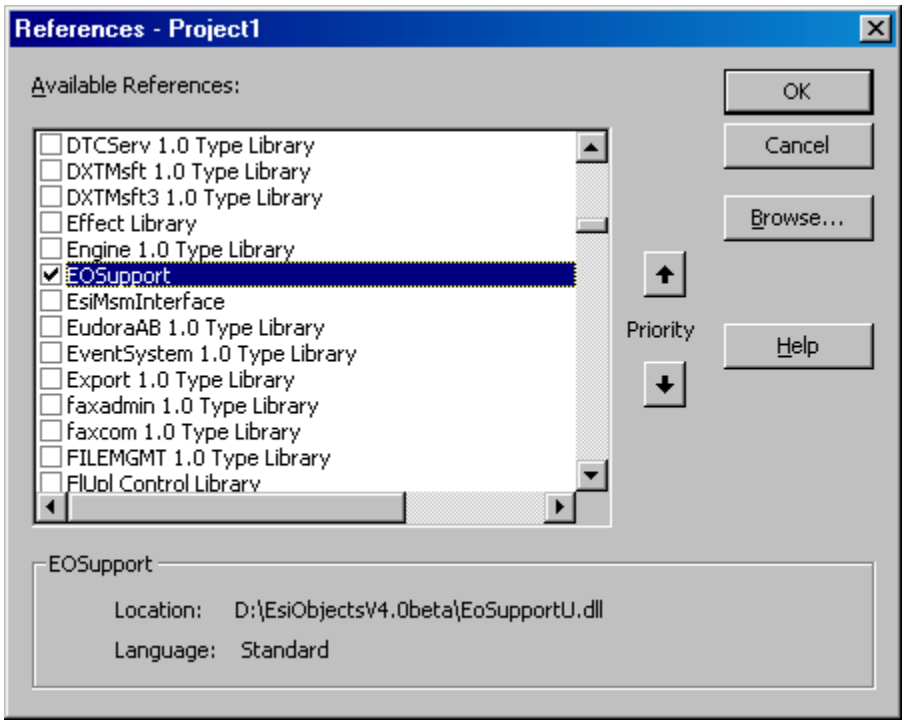
The TCP Bridge icon should now appear in the VB control toolbox (see picture below.)

The VB Control Toolbox.

The icon at the bottom right is for the TCP Bridge.

Updating the References Dialog to Include EOSupport

If you want to make use of the bulk transfer objects supplied in the control, you will need to add the EOSupport to the project references.



The Project References Dialog

Follow these instruction to add EOSupport to the project:

4. From the **Project** menu, select the **References** option.
5. Check the “EOSupport” item as shown above.
6. Select “OK”

The Object Browser (under the View menu) should now list the EOSupport library.

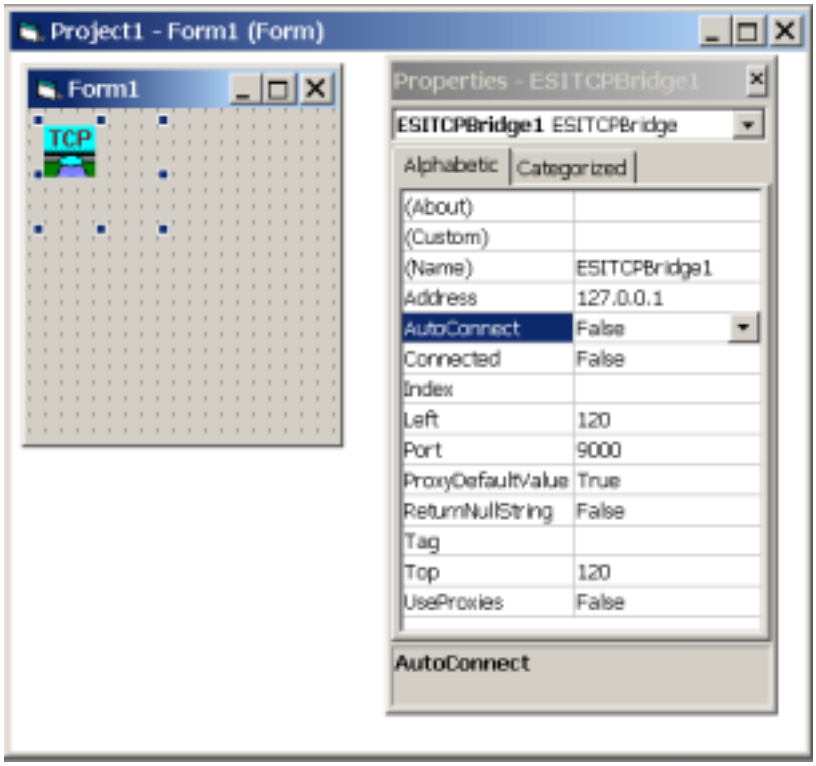
Adding a Bridge to Your Project

Now that you have the control available to you on the toolbox, placing it within your project is as simple as adding any control to your form. The following outlines the steps to add the bridge to your project.

1. Select the Control Icon
2. Add to your Main Form
3. Configure the properties of the Bridge
4. Make sure that Object Proxies are enabled (the *UseProxies* property.).

Since the control is hidden at runtime it doesn't matter where it is placed on the form.

Configuring the properties of the control is an important step. Below is a picture of a bridge control that has been added to a VB form. The Properties dialog allows you to configure the control's properties.



Bridge Control

The TCP Bridge placed on a VB form and selected, showing the Properties dialog.

- Name – *the name of the control.*
- Address – *the TCP/IP address of the EsiObjects server.*
- AutoConnect – *setting this to TRUE will cause the control to automatically connect to the EsiObjects server when the form is loaded. The control will attempt to connect to the EsiObjects listener running on the specified* Address *and* Port. *A setting of FALSE will not connect the bridge to the server automatically.*
- Connected – *a read-only property. TRUE indicates that the bridge is connected to the server.*

- Port – *the TCP/IP port where an EsiObjects listener is running waiting to accept a connection*
- ProxyDefaultValue **–** *if set to true proxy objects will have the OID string of the object which they proxy as their default value.*
- ReturnNullString **–** *if set to true a null string on the server will be returned as an empty variant. If set to false a Null variant is returned.*
- UseProxies – *if set to TRUE, you can use the VB proxy objects to map directly to objects on the server. Generally this should be set to TRUE unless you are using VB code that was written under an older version of the bridge that did not make use of proxies. In that case, communications with server objects was done via a functional API.*

  Configure the above properties according to your environment.

Connecting to the Server

Once you have properly installed and configured the control within your project, you should determine how the control will connect with the EsiObjects server.

The EsiObjects TCP Listener

The *EsiObjects listener* is a process that is started up within the EsiObjects environment on a specific port. This process then listens on the port for incoming connections and processes requests.

Using Cache

Within Cache environments, the server is started up using the following command:

```
DO START^VESOTCSV(port_number)
```

This will start the listener on the specified port. Connections from clients can now be made to this *port_number*. Cache automatically accepts multiple connections on this port and spawns separate processes off for each connection.

Using DSM

Within DSM environments, there are two types of listeners that can be started. In one case, the listener is started on a single port and waits for a single connection. Only one connection can be made on that port, and once the client disconnects from the port, the listener process exits. The other listener is called a *redirector* because it starts on a single port, listening for incoming connections, but then redirects clients with a listener on another port. That new listener then communicates directly with the client until the client closes the connection.

`DO START^VESOTCSV(port_number)`

The command above, when executed in a DSM EsiObjects server environment will cause the listener to listen for a single connection on the specified *port_number*. No other connections are allowed on this port once a client is connected. Once the client disconnects, this process exits back to the M prompt.

`DO START^VESOTCPR(port_number,begin,end)`

The command above starts a redirector on the specified *port_number*. Incoming client connections on this port will be redirected to listeners beginning with port *begin* up to port *end*.

Automatic Connection

When the *AutoConnect* property of the ESITCPBridge is TRUE, then the bridge will automatically attempt to connect when the control is loaded. The connection is made using the *Address* and *Port* properties. An EsiObjects TCP Listener must be running on the TCP/IP address and port specified.

When to Use

Automatic Connection is useful for those applications where the EsiObjects Server Address and Port will not change.

Explicit Connection

Generally, users of the bridge will connect to the server explicitly. Often an application may lookup local information about what server and port to connect to from its configuration information and then explicitly connect to that server. Some applications may query the user for this information. Explicit connections can be made to a server via the **Connect** or **ConnectTo** method on the bridge. Both methods return TRUE or FALSE to indicate whether or not the connection was successful.

When to Use

Explicitly connect to the server when connection information is not constant for the application.

Connect

The **Connect()** method uses the current state of the *Address* and *Port* properties for making the connection to a server. These properties can be set into the control at design time using the control's property sheet, or during runtime is in the following example.

Example

```
Private Sub Form_Load()
ESITCPBridge1.Address=”appsrv4.esitechnology.com”
ESITCPBridge1.Port=9000
If ESITCPBridge1.Connect = False then Exit Sub
.
.
.
End Sub
```

ConnectTo

The **ConnectTo()** method takes two input parameters: a string Address and an integer Port which specify the address and port to connect to. These inputs do not change the *Address* and *Port* properties of the bridge control.

Example

```
If ESITCPBridge1.ConnectTo(”127.0.0.1”, 9000) = False then Exit Sub
```

Verifying the Connection

There are two ways to verify that a connection has properly been made to the server.

1. Connected Property – checking this property will return TRUE if a connection exists.
2. Connect Event – the Broker generates a Connect event that indicates a connection has been made.

What to Do if the Connection Fails?

If your connection fails there are several things you should check.

1. Ensure that *Address* and *Port* you are specifying, either in **ConnectTo()** method or in the control's properties, are correct.
2. Ensure that the Server Listener is running at the requested port.
3. Ensure that the TCP/IP address is reachable from your location. (Try a “ping” for starters.)
4. Ensure that the Server Listener has enough connections available.
5. Check with you system administrator for firewalls that may be preventing connections.

Disconnecting from the Server

Disconnecting from the server can happen in one of two ways: explicitly by the programmer or automatically when the control is unloaded.

Explicitly Disconnecting

It is possible to explicitly terminate a connection to a server by using the **Disconnect()** method.

Example

```
ESITCPBridge1.Disconnect()
```

Automatic Disconnection

When a form containing a TCP Bridge is unloaded, the bridge will disconnect prior to being unloaded. Under some error conditions the TCP Bridge was automatically disconnect from the server.

The effects of Disconnection

- The Broker will not longer dispatch messages or events.
- Proxies will not longer function.
- A Disconnect event will be thrown if the TCP Bridge was connected before.

The Broker Object

Once a connection is made to the server, all subsequent communications with EsiObjects is done via the *Broker object*. A handle to this object is obtained by invoking the **Broker** method on the bridge. This operation returns the handle to the object, which is then used to invoke object services in EsiObjects.

It is convenient to make this object handle a global one in your VB application so it can be accessed from any module. To do this, simply add the following line to the General Section of the VB code:

```
Dim Broker As ESITCPBroker
```

Example

After successful connection to the server, the Broker is obtained in the following example.

```
Broker = ESITCPBridge1.Broker
```

Using LookupObject

Using the Broker object, one of the first tasks may be locating objects on the server. The **LookupObject()** is one mechanism for locating persistent and system

objects on the server. This method takes one input parameter that is the name of the object (see below for types of named objects). It returns either an empty string if no such object was found, or a Variant/Object containing the handle to the server object associated with the specified name.

Locating System Variables

The table below lists the system objects that the **LookupObject** service may find. The names in the table are not case sensitive.

System Object	Abbreviation	Description
$ENVIRONMENT	$ENV	The Environment object associated with this connections
$LIBRARY	$LIB	The default Library
$LIBRARYLIST		The List of all Class Libraries
$SYSPOOL		The System Name Pool

Example

```
‘Find $ENV
Dim EnvObj as Object
Set EnvObj=Broker.LookupObject(“$Env”)
```

Locating Class Objects

The **LookupObject** service may be used to find the Class object associated with a class name. When looking up the class the name used should be the full class name prefixed with an underscore. The standard format for nested class names can be used.

Note: When getting an object from an operation such as **LookupObject()** you can Dim the variable as an Object or Variant. If you do not Dim the variable, the data type is set to Variant/Object.

Examples

```
‘Find the Class Base$Set
Dim SetCls as Object
Set SetCls=Broker.LookupObject(“_Base$Set”)
‘ SetCls data type is Object
‘Find the Nested Class ESI$MyClass>Nested1>Nested2
Set NestCls=Broker.LookupObject(“_ESI$MyClass>Nested1>Nested2”)
‘ NestCls data type is Variant/Object
```

Locating an O% Name

LookupObject may also be used to find named objects in the current default domain. When coding in EsiObjects such names are prefixed with an “O%”. When using the **LookupObject** service the O% should not be used – instead use just the name.

If a name is not found then an empty string will be returned. It is thus possible to check to see if name is defined by checking against the empty string.

Examples

```
‘Check if an O%Database is defined
If Broker.LookupObject(“Database”) = “” Then ...
‘Code to deal with the undefined Database
Else
‘The Database is defined - let’s get it
Set DB = Broker.LookupObject(“Database”)
EndIf
```

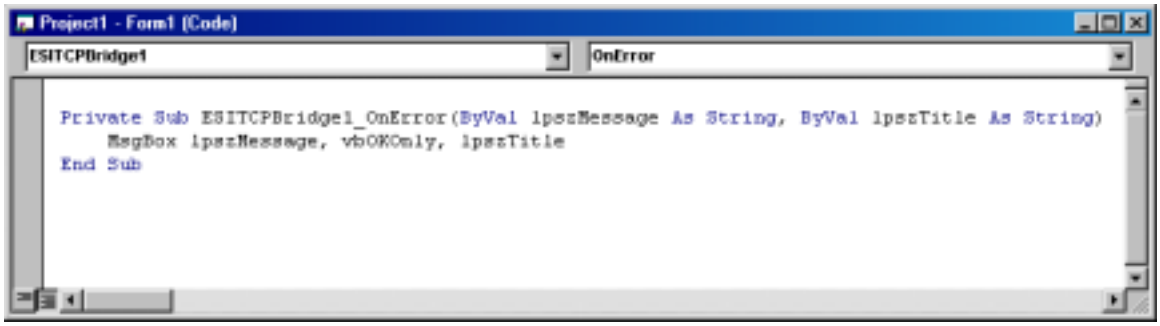
Trapping Server Errors

What are Server Errors?

If an error occurs while processing a service on the server the bridge will receive an event with the error information. By defining code to handle this event you may be informed of server problems.

Adding an Event Handler to Trap Server Errors

1. From the **View** menu, select the **Code** option.
2. Select the TCP bridge object from the Object dropdown list.
3. Select the **OnError** procedure from the Procedure dropdown list.
4. Write your event handler.



The code editor window for the OnError procedure

Creating and Destroying Objects (Lifespan Services)

Lifespan services are those services used to create and destroy objects on the server. They allow the client to directly create and destroy objects. This section of the guide deals with creation and destruction of objects using the bridge. The Broker object is used to create and destroy objects. There are two methods supplied with the Broker for creating objects: **SimpleCreateObject()** and **CreateObject()**. The method **DestroyObject()** is used for object destruction.

In EsiObjects, the format of the Create command is as follows:

```
Create var=ClassName(param1,param2,…):(keyword=value,…):(property=value,…)
```

Recall that the *params* are those parameters that are passed into the CREATE method in the Factory interface.

SimpleCreateObject allows for specifying positional parameters, only the SHARE keyword, and no property assignments. **CreateObject** allows you to make full use of the EsiObjects construction syntax – both positional and named parameters, most creation keywords, and property assignments.

SimpleCreateObject

Syntax: SimpleCreateObject(classname, flags, param1, param2,…**)**

This method handles simple cases of object construction. The method returns an Object. Only *classname* and *flags* are required. Use this method if you wish to create objects without using named parameters. It does allow for creating shared objects. In summary, SimpleCreateObject:

1. Is used to create objects with out special construction requirements.
2. Supports creation of shared objects.
3. Does not support specifying FIXED or BASE locations for the object.
4. Supports positional parameter passing only.
5. Does not support specifying properties for the construction of the object.

Use CreateObject() if other creation features are needed, such as named parameters or property assignments.

The method requires two input parameters:

1. The class name (string) for the object you want to create. You should use the fully qualified class name, which consists of the class library name along with the class name, e.g. “Base$Array”.
2. A flag (long) indicating whether the object is to be a child (0) or a shared object (1).

All input parameters after this are for positional parameters for the Factory::CREATE method of the class. A comma should separate each

parameter. For more information on SimpleCreateObject() please refer to the reference section below.

Examples

The following example shows the creation of two objects. Note that the return value from the SimpleCreateObject is a data type of Object. Thus you should "Dim" the handles as Objects.

```
Dim Query As Object
‘Create a temporary object of the hypothetical class SuperBase$Query
‘This class requires three creation parameters
‘   The database to use for the Query
‘   The UserName
‘   The Encrypted Password
Set Query =Broker.SimpleCreateObject(“Superbase$Query”,0, _
”MainDatabase”, _
”Jones, Fred C”, _
”LKE028833HGX2”)
‘When we are done with the object we should destroy it.
‘Create a Shared object and tell a security tracker object about it.
Dim Person As Object
Set Person=Broker.SimpleCreateObject(“Corp$Person”,1,”Jones, Fred C”)
SecurityTracker.RegisterPerson Person
```

CreateObject

Syntax: CreateObject(classname, flags, PositionalParms, NamedParams, Options, Properties**)**

This method allows objects to be created using a broad range of functionality. It supports almost every aspect of the EsiObjects Create command. The method returns an Object. In addition to the functionality provided by SimpleCreateObject() this method supports:

1. Most creation keywords (see the reference section below for more information on the keywords that are supported.)
2. Named parameter passing.
3. Creation-time property assignment.

Like SimpleCreateObject(), only the first two parameters are required: the class name and the flags. The other parameters are optional, they be omitted or an empty collection passed in their place. These other parameters make use of the data types supplied in the EOSupport library provided in the bridge. Objects of these data types are for transfer of collections of data to and from a server – also know as Bulk Transfer Objects. We describe these objects in detail below. For now, the example code shows their use in creating an object and passing in positional and named parameters, creation keywords, etc.

Positional parameters are passed as an EsiList.

Named parameters are passed as an EsiNvList. The Value of each element in the list is the value of the parameter, while the Name of the element is used as the parameter name.

Creation Options are passed as an EsiNvList. The name of each element is the creation keyword (case sensitive), while the value of the element is the value assigned to that creation keyword.

Creation properties are passed as an EsiNvList. The name of each element is the name of the property, while the value of the element is the value to be assigned to that property.

For more information on CreateObject() please refer to the reference section below.

Examples

```
Dim OrderServer As Object
Dim Parameters As New EsiList
Dim Options As New EsiNvList
Dim NamedParams As New EsiNvList
Dim Properties As New EsiNvList
‘We are going to create an object of the hypothetical class
“Local$OrderServer”
‘We want to create this class in the default domain,
‘and give it a name of OrderServer.
‘To do this will need to define two Create options
Options.SetAt “Domain”, ””             ‘Use the default domain
Options.SetAt “Name”, ”OrderServer”    ‘The Name=“OrderServer”
‘In order to Create an Object Server we specific an few parameters
‘The first two params may be passed positionally, so we set them
‘in the Parameters Object
Parameters.Dimension=2 ‘Set the size of the parameter object – 2
parameters
Parameters.SetElement 0, ”Jones, Mark”    ‘The User Name –  param 1
Parameters.SetElement 1, ”All” ‘Scope code for what orders to service –
param 2
‘The creation also requires an authentication token, (passed as
NamedParameter)
NamedParams.SetPair –1, ”AToken”, Token    ‘Pass Token we have already
defined.
‘We also want to define a few properties of the object when it is
created
Properties.SetAt “MaxClients”, 10
Properties.SetAt “EnableEvents”, 1
‘Ok we are now ready to Create the Object
Set OrderServer = Broker.CreateObject(“Local$OrderServer”,0, _
Parameters,NamedParams, _
Options,Properties)
‘Example of the same creation without properties
```

```
Set OrderServer = Broker.CreateObject(“Local$OrderServer”,0, _
Parameters,NamedParams,Options)
‘Example without named parameters or properties
Set OrderServer = Broker.CreateObject(“Local$OrderServer”,0, _
Parameters,,Options)
```

DestroyObject

Syntax: DestroyObject(*objectId*)

This method destroys the server object corresponding to the *objectId*. It is equivalent to the Destroy command in EsiObjects. Keep in mind that in EsiObjects, when using Destroy the object may not always die. The object may have their reference count decremented instead, or they may reject the attempt altogether. The method returns a Boolean to indicate whether the object was successfully destroyed (or had its reference count decremented), or whether the attempt was rejected.

Examples

```
‘ Destroy the OrderServer object created in the previous example.
Broker.DestroyObject OrderServer
‘ Example of capturing the return value of the DestroyObject
Dim test As Boolean
test = Broker.DestroyObject(OrderServer)
```

Using the Bridge with Proxies

This section describes the concepts and uses of Proxies in Visual Basic and the TCP Bridge. In the examples above, we created objects on the server and *Proxy Objects* within VB. For example, the OrderServer object created above is a Proxy object that maps to the actual object within the EsiObjects server. This assumes that the *UseProxies* property on the Bridge is set to TRUE. Invoking operations on the proxy object will cause the corresponding operation to be performed on the actual server object in EsiObjects. The bridge handles this mapping and routing transparently.

Concepts

Proxies map to Server Objects. Each proxy supports all the methods, properties, and relationships that belong to its corresponding server object. They do this by forwarding all service requests to the server object. The proxies do not act locally. Also, keep in mind that proxies are generic – all validation occurs on the server. Thus if you invoke an operation on a proxy that is not implemented in the server object, an error will occur on the server indicating that the specified service could not be found.

The *UseProxies* property must be set to TRUE on the bridge in order for the bridge to generate proxy objects.

Default Value

The default value of a Proxy is the EsiObjects OID for the object to which it is a proxy.

Example

```
Set Env=Broker.LookupObject(“$ENV”)
EnvOID = Env
‘EnvOID should now be the OID of the Environment Object
```

Invoking Object Services

In most cases invoking a method, property or relationship service on a proxy object is similar to invoking a service on any VB object:
`object.servicename`

This syntax is used for invoking a service in the server object’s Primary interface and passing no parameters. Below we describe how to invoke services in other interfaces and how to pass parameters.

Primary Interface

The services from the primary interface of an object are available directly on an object proxy by simply using their name.

Format of the Name

ServiceName

Example

In the example above, we invoked **LookupObject** to get a handle to the ESI$Environment object in EsiObjects. Env now is a Proxy object mapped to that server object. The environment object has a property in the Primary interface called *FullName* that returns the name of the environment object. This example invokes that property to get its value.

```
Name=EnvObj.FullName
```

Unless the service returns an object, the return values from invoking services in the proxy are *string* data types. If the return value is an object id, the return value is a *Variant/Object*.

Other Interfaces

To explicitly access a service on an interface other then the Primary interface, the name of the interface must also be included. Preceding the service name with the interface name, separated by an underscore, forms the name.

Format of the Name

Interface_ServiceName

Example

This example shows how to execute the Upgrade method in the Factory interface of the environment object. No return value is captured.

```
Env.Factory_Upgrade
```

Using Parameters

Many services expect input parameters. EsiObjects allows for both positional and named parameter passing into a method. The bridge also allows for both.

Positional Parameters

Positional parameters are mapped to the input specification parameters in the order they are passed.

Example

The Assert method in ESI$Environment displays a message box on the client. Several inputs can be specified for this method including the text of the message and the title. The following example shows an invocation of the Assert message passing in these two parameters positionally. The first parameter is the message text, the second is the title.

```
Env.Assert “Hello World ”, “EsiObjects Bridge”
```

EsiObjects Assert Message

Named Parameters

The proxy also supports sending named parameters using the syntax *Name*:=*value*.

Example

```
Env.Assert Text:=”Hello World Again”, Title:=”EsiObjects Bridge”, _
Icon:=”Stop”, Buttons:=”OK”
```

Proxy Assert Message

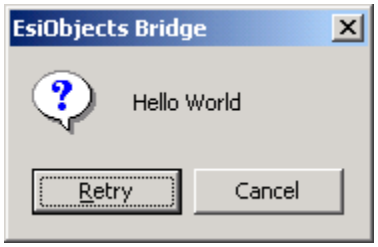
Empty Parameters

When passing parameters in a positional manner it is possible to use an empty parameter. This specifies that this positional parameter is empty. The server will receive a null sting for the parameter value.

Example

In this example, parameters three and four are passed as nulls into the server method, parameter five is for the icon to be displayed on the message box.

```
Env.Assert “Hello World ”, “EsiObjects Bridge”, , , “Question”
```



Assert Message using Parameters

Mixed Parameter Usage

It is possible to use mix the two types of parameter passing. **Note: All positional parameters must occur first!** Of course, empty positional parameters may be used as well.

Example

Instead of passing nulls for parameters three and four as in the example above, this example uses a mixture to pass positional parameters one and two, then specifies the Icon named parameter.

```
Env.Assert “Hello World ”, “EsiObjects Bridge”, Icon:=“Question”
```

Bulk Data Transfer Mechanisms

The TCP Bridge contains a number of data types that allow for the transfer of collections or large amounts of data in bulk to and from the server. As we have seen in the sections above, the bridge itself uses several of these objects. For example, in the **CreateObject()** service. To pass a collection of parameters on the object creation, an EsiList is used. Named parameters are passed in an EsiNVList.

This section deals with what these objects are and how to use them.

Concepts

Below is a table of the bulk transfer objects (BTO) supplied in the EOSupport library.

Bulk Transfer Types

Name	EsiObjects Equivalent	Description	Data by Value?	Collection?	Server Creatable?
EsiList	ESI$List	A list of items	Yes	Yes	Yes
EsiNVList	ESI$NVList	A list of items with Names	Yes	Yes	Yes
EsiTable	ESI$Table	A multicolumn table	Yes	Yes	Yes
EsiText	ESI$Text	A Large Text object	Yes	No	Yes
EsiStream	ESI$TCPStream	An IO Stream	No	No	No

The List, NVList, and Table objects are collections into which elements can be inserted, removed and iterated.

Text and Stream are objects to deal with large amounts of string data. All of these objects, except the Stream, can be created on the server and passed to the client. In other words, an ESI$List object created in EsiObjects can be returned to the client and used within the client as an EsiList object. Conversely, if the client creates an EsiList object and passes it into the server in a method parameter, the resultant object in the server is an ESI$List object.

Transfer Dynamics

When a BTO object uses “data by value” parameter passing (Streams do not use data by value), when that object is passed as an argument to a service, or used as a return value, the entire contents of that collection is transferred to the other side. In the case of a BTO object used as an argument of a call from the client to the

server, the contents of the object will be transferred to the server in total, and at the conclusion of the call the new contents will be transferred back to the client.

Common Conventions

Index Basis

All structures have elements from zero (0) to their Dimension minus one(1). Thus if you set the *Dimension* property of an EsiList to 10, the indexes go from zero to nine.

Effects of Dimension

An attempt to set an element outside of the dimensioned bounds of a collection will result in an error. This means that all collections should be dimensioned prior to being filed. The only exception is when an append operation is being performed, which will cause the collection to grow.

Appending Items

Setting an item at position negative one (-1) will cause the item to be placed at the end of the collection and the dimension of the collection to be increased.

Common Error Handing

All BTO objects that support “Data by Value” support a set of common error handling facilities. These facilities allow error information to be associated with the collection directly. These facilities might be used by methods that validate the contents on collections.

See the section Common Error Methods below for details on these facilities.

Creating BTO objects in Visual Basic

If you need a BTO object in Visual Basic, there are two mechanisms for creating one: using the **CreateObject()** service, or using the Dim command.

Using CreateObject

```
Dim Table as EsiTable
Set Table = CreateObject(“Esi.Table”)
```

Using the Dim Command

```
Dim Table as new EsiTable
```

Between Bridges

Unlike proxies which represent objects in the server that exist only in the context of the single bridge connection (unless it is a shared object), BTO objects can be

used between different bridges. So if your project contains more than one bridge, both can use a BTO object. This is because the entire contents of the object are transmitted to the client and server and thus can be shared by different bridges.

Common Error Methods

Below are detailed the common error methods supported by all BTO object that support “Data by Value”. They provide a set of common facilities for associating error information with a BTO object.

Methods

GetErrorId

Returns the error id number.

Signature

Function: GetErrorId() As Long

Returns: The error id number.

GetErrorItem

Returns the item associated with the error.

Signature

Function: GetErrorItem() As String

Returns: The item associated with the error.

GetErrorText

Gets the text of the error.

Signature

Function: GetErrorText() As String

Returns: The text of the error.

SetErrorId

Used for setting an error id number that can be used to identify an error.

Signature

Function: SetErrorId(Id As Long)

Parameters: Id – the error id number.

SetErrorItem

Used to set an item into the error data that was associated with the error. For example, if a particular item in the collection were invalid, the error item would be set to that item.

Signature

Sub SetErrorItem(Item As String)

Parameters: Item – the item associated with the error.

SetErrorText

Sets the text of the error message.

Signature

Sub SetErrorText(Text As String)

Parameters: Text – the text of the error message.

Bulk Transfer Object (BTO) Reference

EsiList

Description

An EsiList is a simple singleton list of items.

Methods

ClearAll

Clears the contents of the List.

Signature

Sub ClearAll()

GetElement

Gets an element in the List.

Signature

Function: GetElement(Item As Long) As Variant

Returns: The Value of the requested element

Parameters: Item - The zero based index of the item to retrieve.

SetElement

Sets an element into the List. If an item is already stored at the specific index, that item will be overwritten with the new value.

Signature

Function: SetElement(Item As Long, Value As Variant) As Long

Returns: The position the element was set at.

Parameters:

Item - A zero based index of the item in the list. A value of negative one (-1) indicates that the element should be append to the end of the list.

Value - The value of the element.

Properties

Dimension

Used to set and retrieve the number of cells in the list. The cells may or may not have any elements stored in them.

Signature

Dimension As Long

Returns: The number of cells in the list.

EsiObjects Equivalent

ESI$List

Within EsiObjects, a BTO object can be created from the ESI$List class. Objects created within EsiObjects of this type can be returned to the client and used like a client EsiList.

Example

```
‘Define the List
Dim theList as New EsiList
‘Set the size of the List via the Dimension property
theList.Dimension=12
‘Load the List with some data
For Item = 0 To theList.Dimension-1
     value = “Item: “+ Str(Item)
     theList.SetElement Item, value
Next
‘Append a few items into the List via the SetElement using –1 index
theList.SetElement -1, ”Extra Item 1”
```

```
theList.SetElement -1, ”Extra Item 2”
‘Fill a ListBox with items from the List
For Item = 0 To theList.Dimension-1
     value = theList.GetElement(Item)
     Listbox1.AddItem value
Next
```

EsiNVList

Description

The EsiNVList provides a list of named items. Each item in the list may be given a name – also called a “key”. Names need not be unique. An index based on name is maintained which allows items to be retrieved by name. An NVList object can also be used like a Map that maps names to values.

Methods

ClearAll

Empties the EsiNVList of all contents.

Signature

Sub ClearAll()

DoesKeyExist

Determines is a specific name exists in the NVList

Signature

Function: DoesKeyExist(Name As String) As Long

Returns: Non zero if the Name is defined in the NVList

Parameters: Name - the name to look for.

GetCell

Returns a *NamedCell* object (described below) that is a copy of the element stored at the specified index within the collection.

Signature

Function: GetCell(Item As Long) As NamedCell

Returns: A NamedCell object with a copy of the information located at this element.

Parameters: Item - the zero based index of the element being requested.

GetName

Gets the Name (the key) of an element stored at the specified index.

Signature

Function: GetName(Item As Long) As String

Returns: The Name field of the element stored at the specified index.

Parameters: Item - the zero based index of the element.

GetValue

Gets the Value field of an element stored at the specified index.

Signature

Function: GetValue(Item As Long) As Variant

Returns: The Value field of the element stored at the specific index.

Parameters: Item – the zero based index of the element.

Lookup

Similar to **GetValue()** except that it finds the Value associated with a specified Name instead of an index. The method throws an error is the name is not found.

Signature

Function: Lookup(Name As String) As Variant

Returns: The Value associated with the specified Name.

Parameters: Name – the name of the element to lookup.

RemoveKey

Deletes the element associated with the specified Name. The size of the NVList is unchanged – the cell occupied by the removed element is blank. If there exists in the collection more than one element with the same Name, only the most recently inserted element with that Name will be cleared.

Signature

Sub RemoveKey(Name As String)

Parameters: Name – the Name (key) of the element to remove.

SetAt

Sets the Value field associated with the specified Name. If the Name is not present in the NVList, a new element will be created. If there exists more than

one element in the collection of the Name, only the most recently inserted element of that Name will be modified.

Signature

Sub SetAt(Name As String, Value As Variant)

Parameters:

Name – the Name of the element into which we are setting a new Value.

Value – the new Value of the element associated with the specified Name.

SetCell

Sets an element using a *NamedCell* object (described below) at the specified index. Any previous contents of the cell at that index will be overwritten.

Signature

Sub SetCell(Item As Long, Cell As Object)

Parameters:

Item – the zero-based index number of the element to set. A value of –1 will cause a new element to be added to the end of the NVList.

Cell – a *NamedCell* object with the Name/Value pair for this element. The information will be copied from this object into the cell at the specified index.

SetName

Sets the Name field of an element at the specified index. Any previous name stored at that index will be overwritten.

Signature

Sub SetName(Item As Long, Name As String)

Parameters:

Item – the zero-based index number of the element to set. A value of –1 will cause a new element to be added to the end of the NVList. In this case, the Name field will be set and the Value field will be blank.

Name – the new Name for the element.

SetPair

Similar to **SetCell()**, this method sets the Name/Value pair of an element at a specified index, except the Name and Value are passed as input parameters instead of within a *NamedCell* object. Any previous contents of the cell at that index are overwritten.

Signature

Sub SetPair(Item As Long, Name As String, Value As Variant)

Parameters:

Item – the zero-based index number of the element to set. A value of –1 will cause a new element to be added to the end of the NVList.

Name – the new Name for the element.

Value – the new Value for the element.

SetValue

Sets the Value of an element at the specified index. Any previous value stored at that location would be overwritten.

Signature

Sub SetValue(Item As Long, Value As Variant)

Parameters:

Item – the zero-base index number of the element to set. A value of –1 will cause a new element to be added to the end of the NVList.

Value – the new Value for the element.

Properties

Dimension

Used to set and retrieve the number of cells in the list. The cells may or may not have any elements stored in them.

Signature

Dimension As Long

Returns: The number of cells in the list.

EsiObjects Equivalent

ESI$NVList

Within EsiObjects, a BTO object can be created from the ESI$NVList class. Objects created within EsiObjects of this type can be returned to the client and used like a client EsiNVList.

The NamedCell Object

This object mimics a cell in an NVList. It has two fields: Name and Value. Methods on the NVList can make use of a NamedCell object to set and retrieve an element from an NVList.

Properties

Name

The *Name* field of the cell.

Signature

Name As String

Value

The *Value* field of the cell.

Signature

Value As Variant

Examples:

```
‘Example of Using an EsiNVList
‘
Dim NvList As new EsiNVList
‘Allocate space for five (5) items
‘
NvList.Dimension=5
‘Set data into the NVList by position in various ways
NvList.SetPair 0, ”First”, 1
NvList.SetPair 1, ”Second”, 2
NvList.SetValue 2, 3   ‘ sets the Value field of the 3rd element to “3”
NvList.SetName 2, ”Third”   ‘ sets the Name field of the 3rd element
‘ Set the 4th element using a NamedCell object
Dim ACell As new NamedCell
ACell.Value=4
ACell.Name=”Forth”
‘ set the cell into the fourth cell
NvList.SetCell 3, ACell
‘ reuse the same cell to set the 5th element
ACell.Name=”Fifth”
ACell.Valyue=5
NvList.SetCell 4, Acell
‘Append an element onto the NVList
NvList.SetPair -1, ”Sixth”, 6
‘Set using a key that does not exist yet
NvList.SetAt “Seventh”, 7
```

```
‘Lookup some Values in various ways
Temp = NvList.GetValue(3) ‘Temp Should = 4
Temp = NvList.Lookup(“Second”) ‘Temp Should = 2
Set ACell = NvList.GetCell(2)
Temp = ACell.Name ‘Temp should = “Third”
```

Issues with Names (keys) in an NVList

Multiple elements with the same Name (key) value can provide unpredictable results. Generally, the most recent definition of the key will be the one used for operations such as Lookup, SetAt, etc. Because of this behavior, EsiNVList is intended to be used as a map for unique names to values and should be used for that purpose.

EsiTable

Description

An EsiTable is a collection of data organized by rows & columns. It is optimized for row based operations more than column based lookups, etc.

To be useful the Table must always be dimensioned, although it is legal to specify zero for one of the two dimensions. Rows and columns are zero-based.

Methods

AddRow

Adds a row to the table. The row is passed as an EsiList object. Each element in the list is placed in a column along that row.

Signature

Function: AddRow(List As EsiList) As Long

Returns: The zero-base index of the row that was added

Parameters: List – an EsiList object with the contents of the new row. If the EsiList has more elements then there are defined columns, the additional elements will be ignored. If the list is shorter then the number of columns, missing columns will be empty.

Example

```
Dim Table As New ESITable
‘Dimension the Table as 0 Rows of 4 Columns
Table.SetDimension 0, 4
‘ add elements to a list which will become columns in a row
Dim List As New EsiList
List.Dimension=4
For I = 0 to 3
```

```
    Data=”Data “+Str(I)
    List.SetElement I, Data
Next
NewRow=Table.AddRow(List)
```

AddRowV

Similar to **AddRow()** this method adds a row to the table, but the column values are passed in via explicit input parameters.

Signature

Function: AddRowV(ParamArray VariableArgs() As Variant) As Long

Returns: The zero-based index of the row that was added.

Parameters: ParamArray – a variable number of arguments, all of which are Variants.

Example

Special note: As of this writing, a bug exists in the implementation of the **AddRowV()** service. The workaround is to dimension the variable for the Table as an Object and then use the *New* command (as shown below.)

```
‘Dim Table As New EsiTable ‘ AddRowV will not work
Dim Table As Object           ‘ Workaround to get it to work
      Set Table = New EsiTable
‘Dimension the Table as 0 Rows of 4 Columns
Table.SetDimension 0,4
NewRow=Table.AddRowV(“Data1”,”Data2”,”Data3”,”Data4”)
NewRow=Table.AddRowV(“Data5”,”Data6”,”Data7”,”Data8”)
```

ClearAll

Clears the contents of the table.

Signature

Sub ClearAll()

Columns

Returns the number of Columns in the Table

Signature

Function Columns() As Long

Returns: The number of columns in the Table.

GetCell

Returns the value stored in a cell in the table at a specified row and column.

Signature

Function GetCell(Row As Long Column As Long) As Variant

Returns: The value stored in the cell.

Parameters:

Row – the zero based index of the row.

Column – the zero based index of the column.

GetColumn

Returns an EsiList of the content of a column in the table from all rows. This function is much slower than **GetRow()** since the table is optimized for row access.

Signature

Function GetColumn(Column As Long) As EsiList

Returns: An EsiList object with the contents of the requested column from all rows.

Parameters: Column – the zero-based index of the column to retrieve.

GetDimension

Returns the dimensions of the table. The return value is a *string* in the format: “Rows,Columns”.

Signature

Function GetDimension() As String

Returns: A string representation of the dimensions of the table. Formatted as “Rows,Columns”.

GetRow

Returns an EsiList of the content of the specified row in the table.

Signature

Function GetRow(Row As Long) As ESIList

Returns: An EsiList with the contents of the row. Each column value occupying one cell in the list.

Parameters: Row – the zero-based index of the row to retrieve.

Rows

Returns the number of Rowss in the Table

Signature

Function Rows() As Long

Returns: The number of rows in the Table.

SetCell

Sets the value of a cell in the table at the specified row and column.

Signature

Sub SetCell(Row As Long, Column As Long, Value As Variant)

Parameters:

Row – the zero-based index of the row.

Column – the zero-based index of the column.

Value – the value to the set the cell.

SetColumn

Sets an entire column of data to the contents of an EsiList. Each element in the list is set into a row of the table at the specified column. This operation is much slower than inserting data via a SetRow(), since the table is optimized for Row operations.

Signature

Function SetColumn(Column As Long, List As ESIList) As Long

Returns: The zero-based index of the column that was set.

Parameters:

Column – the zero-based index of the column to set. A value of negative one (-1) will append a new column.

List – an EsiList with the contents of the new columns. Any entries in the list beyond the number of rows in the table are ignored. If the elements in the List are shorter than the rows in the table, then the remaining items are filed as empty variants. Thus all column values are overwritten.

SetColumnV

Similar to **SetColumn()** this method sets an entire column of data in the table except that the column values are passed in explicitly via input parameters.

Signature

Function SetColumnV(Column As Long, ParamArray VariableArgs() As Variant) As Long

Returns: The zero-based index of the column that was set.

Parameters:

Column – the zero-based index of the column to set. A value of negative one (-1) will append a new column.

ParamArray – a variable number of arguments, all of which are Variants. Each value is placed in a row at the specified column.

SetDimension

Sets the dimensions of the table. Either the rows or columns may be dimensioned to zero (0).

Signature

Sub SetDimension(Rows As Long, Columns As Long)

Parameters:

Rows – the number rows the table should have.

Columns – the number of columns the table should have.

SetRow

Sets the contents of a specified row to the values found in an EsiList. Any values already stored at that row are overwritten.

Signature

Function SetRow(Row As Long, List As ESIList) As Long

Returns: The zero-based index of the row that was set.

Parameters:

Row – the zero-based index of the row to set. A value of negative one (-1) will append a new row to the table.

List – an EsiList containing the data to be set in the row. If the number of entries in the List is longer than the number of columns, the additional items will be

ignored. If the List is shorter, then the additional columns in that row will remain unchanged.

SetRowV

Similar to **SetRow()** this method sets the contents of a specified row in the table, except the column values are passed in explicitly via input parameters.

Signature

Function SetRowV(Row As Long, ParamArray VariableArgs() As Variant) As Long

Returns: The zero-base index of the row that was set.

Parameters:

Row – the zero-based index of the row to set. A value of negative one (-1) will append a new row.

ParamArray – a variable number of arguments, all of which are Variants.

Example

```
Dim Table As New EsiTable
‘Dimension the Table as 10 Rows of 4 Columns
Table.SetDimension 10, 4
‘Update the Data in Row 10
Table.SetRowV 10, “Data1”, ”Data2”, ”Data3”, ”Data4”
```

EsiObjects Equivalent

ESI$Table

Within EsiObjects, a BTO object can be created from the ESI$Table class. Objects created within EsiObjects of this type can be returned to the client and used like a client EsiTable.

Example

```
Dim Table As New EsiTable
'Dimension the Table as 0 Rows of 4 Columns
Table.SetDimension 0, 4
NewRow = Table.AddRowV("Data1", "Data2", "Data3", "Data4")
NewRow = Table.AddRowV("Data5", "Data6", "Data7", "Data8")
'Create a List to use for adding rows to a Table
Dim List As New EsiList
List.Dimension = 4
List.SetElement 0, "ListItem1"
List.SetElement 1, "ListItem2"
List.SetElement 2, "ListItem3"
List.SetElement 3, "ListItem4"
```

```
'Add Ten rows to the table, update the third column to the Value of the
'Loop Index
For I = 1 To 10
    NewRow = Table.AddRow(List)
    Table.SetCell NewRow 3, I
Next
Size = Table.GetDimension() 'Should be "12,4"
```

Unsupported Features

Table headers are not yet supported by the COM version of the Table. However, on the server ESI$Table does implement column headers and they are transported over the wire.

EsiText

Description

The EsiText object supports very long block text. Given the restrictions of most M systems, large blocks of text must be store in a collection of blocks. The EsiText object is the client side representation of the ESI$Text object.

The operations on EsiText facilitate the breaking of the large text object into smaller chunks that can be stored in M.

Access is provided to the text in four ways:

1. As a single text string.
2. By line (As delimited by CR/LF)
3. By sub-string.
4. By text block, given a specific size of block.

The TCP transport will automatically create this type of object on the server side when it receives text that is too large for the M system to handle.

Methods

Append

Appends additional text to the end of an EsiText object.

Signature

Sub Append(Text As String)

Parameters:

Text – the text to append.

BlockCount

Determines the number of blocks of a specified size that are in this text object. Note the final block may actually be smaller than the block size.

Signature

Function BlockCount(Size As Long) As Long

Returns: The number of blocks (of the requested size) that are needed to hold the text.

Parameters:

Size – the size of a block.

Clear

Clears the contents of the text object.

Signature

Sub Clear()

GetBlock

Gets a block of text for a specified block and block size.

Signature

Function GetBlock(Block As Long, BlockSize As Long) As String

Returns: The text for the requested block.

Parameters:

Block – the block to retrieve.

BlockSize – the size of the block.

GetDimension

Returns the number of characters of text in a text object.

Signature

Function GetDimension() As Long

Returns: The number of characters of the text.

GetLine

Gets a specific line if text. Lines are delimited by CR/LF.

Signature

Function GetLine(Line As Long) As Sting

Returns: The line of text.

Parameters:

Line – the zero-based line number to retrieve.

GetSubString

Gets a sub-string within the text, starting from a specified position for a specified length.

Signature

Function GetSubString(Start As Long, Size As Long) As String

Returns: The text starting from the Start position.

Parameters:

Start – the starting character position – zero-based.

Size – the number of characters to retrieve.

GetText

Returns the text as one string.

Signature

Function GetText() As String

Returns: The entire body of text as a string.

LineCount

Gets the number of lines of text. Lines are delimited by CR/LF.

Signature

Function LineCount() As Long

Returns: The number of lines of text.

SetDimension

Sets the size of the Text object. This is useful for pre-allocation of memory.

Signature

Sub SetDimension(Size As Long)

Parameters:

Size – the number of characters expected in the text.

SetText

Sets the entire text of the object to a specified string.

Signature

Sub SetText(Text As String)

Parameters:

Text – the new text to use.

EsiObjects Equivalent

ESI$Text

Within EsiObjects, a BTO object can be created from the ESI$Text class. Objects created within EsiObjects of this type can be returned to the client and used like a client EsiText.

Example

```
‘Create a large text object to send to the Server
Dim theText As New ESIText
‘Define the CR/LF for End of Line
EOL=Chr(13)+Chr(10)
‘Add a 100 Lines to the Text
For A = 1 To 100
   ‘Build the new line, note that EOL is appended.
    Ln = “This is line “+Str(A)+EOL
    theText.Append Ln
Next
‘Determine how many lines there are (Should be 100)
LineCount=theText.LineCount
‘We decide to use a block size to 300 characters
BlockSize=300
‘Determine many 300 character blocks there are
Blocks=theText.BlockCount(300)
‘Loop all of the Blocks
For A = 0 To Blocks-1
    Block=theText.GetBlock(A,BlockSize)
   ‘Do Something with the block
Next
```

EsiStream

Description

The EsiStream object is used when a Client I/O Stream needs to be passed to the server for processing. The current implementation only supports files on the client system and supports *Read* or *Write* mode, but not *Read/Write*.

Information written to the stream by the server is transmitted back to the client and then into the client stream. Likewise a stream *Read* operation on the server causes the client to read information from the stream and transmit it to the server. The EsiStream is an automation implementation of the Microsoft IStream interface.

Methods

Clone

Not yet supported.

Close

Saves any changes and closes the stream. No other operations on the stream are allowed once it is closed.

Signature

Sub Close()

Commit

Saves changes to the stream. The only legal additional operation on a stream that has been committed is **Close()**.

Signature

Function Commit (CommitFlags As Long) As Long

Returns: Success Code.

Parameters:

CommitFlags – see the documentation for IStream for the meaning of these flags.

Name	Value	Supported
Default	0	Yes
Overwrite	1	No

OnlyIfCurrent	2	No
DangerouslyCommitMerelyToDiskCache	4	No
Consolidate	8	No

CopyTo

Not yet supported.

GetSize

Not yet supported.

LockRegion

Not yet supported.

OpenFile

Opens the stream on a requested file. This function is valid on the *Client side only*.

Signature

Function OpenFile(FileName as String, Flags As Long) As Boolean

Returns: TRUE if the file was opened successfully.

Parameters:

FileName – the full path of the file.

Flags – a boolean “Or” of the Flags to use.

modeRead	0	Open in Read Mode
modeWrite	1	Open in Write Mode
modeReadWrite	2	Open in Read/Write Mode
shareCompat	0	Compatible with Sharing
shareExclusive	16	Open with exclusive Access
shareDenyWrite	32	Open Denying Write Access to others
shareDenyRead	48	Open Denying Read Access to others

shareDenyNone	64	Open Denying Nothing to Others
modeNoInherit	128	Do not inherit
modeCreate	4096	Create File if it does not already exist
modeNoTruncate	8192	Prevent Truncation
typeText	16384	Text Mode
typeBinary	32768	Binary Mode

Common Flag combinations

Open for Read	0
Open for Write	1 + 4096 = 4097

Read

Read data from the stream.

Signature

Function Read(Length As Long) As String

Returns: A string of the data read.

Parameters:

Length – the number of characters to read.

ReadLine

Read a line (CR/LF delimited) from the stream.

Signature

Function ReadLine() As String

Returns: The text of the next line, as delimited by CR/LF.

Revert

Not yet supported

Seek

Seek to a position in the Stream, based on a specific origin.

Signature

Function Seek(LowOffset As Long,. HighOffset As Long, Origin As Long) As Long

Returns: Success Code.

Parameters:

LowOffset – the low Word of the offset to seek to.

HighOffset – the high Word of the Offset to seek to.

Origin – code that specifies the origin of the seek.

Start Of File	0
Current Position	1
End Of File	2

SetSize

Sets the size of the stream.

Signature

Function SetSize(LowOffset As Long, HighOffSet As Long) As Long

Returns: Success code.

Parameters:

LowOffset – low word of the Size.

HighOffSet – high Word of the Size.

UnlockRegion

Not yet supported

Write

Writes data to the stream.

Signature

Function Write(Data As String) As Long

Returns: Success Code.

Parameters:

Data – the string data to write out to the stream.

WriteLn

Write a line of string data, terminated by CR/LF.

Signature

Function WriteLn(Line As String) As Long

Returns: Success Code.

Parameters:

Line – the line of string data to write out to the stream.

Properties

AccessTime

Gives the last time the stream was accessed. Depending on the state of the stream, this data may not be available.

Signature

AccessTime As String

CreateTime

The time the Stream was created. May not be available

Signature

CreateTime As String

HighSize

The high word is the size of the stream.

Signature

HighSize As Long

LockType

Not Yet Supported.

Signature

LockType As Long

LowSize

The low word is the size of the stream.

Signature

LowSize As Long

ModifyTime

The time the stream was last modified. May not be available.

Signature

ModifyTime As String

Name

The name of the file associated with the stream. May not be available.

Signature

Name As String

StreamMode

The mode of the stream.

Signature

StreamMode As Long

Possible Values

Group	Storage Mode Identifier	Hex Value	Decimal
Access	STGM_READ	0x00000000L	0
	STGM_WRITE	0x00000001L	1
	STGM_READWRITE	0x00000002L	2
Sharing	STGM_SHARE_DENY_NONE	0x00000040L	64
	STGM_SHARE_DENY_READ	0x00000030L	48
	STGM_SHARE_DENY_WRITE	0x00000020L	32
	STGM_SHARE_EXCLUSIVE	0x00000010L	16
	STGM_PRIORITY	0x00040000L	262144

Creation	STGM_CREATE	0x00001000L	4096
	STGM_CONVERT	0x00020000L	131072
	STGM_FAILIFTHERE	0x00000000L	0
Transactioning	STGM_DIRECT	0x00000000L	0
	STGM_TRANSACTED	0x00010000L	65536
Transactioning Performance	STGM_NOSCRATCH	0x00100000L	1048576
	STGM_NOSNAPSHOT	0x00200000L	2097192
Direct SWMR and Simple	STGM_SIMPLE	0x08000000L	134217728
	STGM_DIRECT_SWMR	0x00400000L	4194384
Delete On Release	STGM_DELETEONRELEASE	0x04000000L	67108864

Success

A success code for the last operation that occurred on the stream.

Signature

Success As Long

Type

The type of stream. The current implementation of stream should return a value of 2, indicating a stream type.

Signature

Type As Long

Unicode

Treat the stream as Unicode text.

Signature

Unicode As Boolean

EsiObjects Equivalent

ESI$TCPStream

This is the only BTO object that cannot be created within EsiObjects and passed back to the client. Only a client can create this object and pass it into the server. Within EsiObjects, an ESI$TCPStream object is created to represent the client side object.

Example

```
Dim WriteStream As New EsiStream
Dim ReadStream As New EsiStream
‘Open File Test.Txt for Write
If WriteStream.OpenFile("Test.Txt", 1 + 4096) Then
            ‘File Test.Txt is Open for Write
            WriteStream.WriteLine "Line 1"
            WriteStream.WriteLine "Line 2"
            WriteStream.Close
End If
‘Open File Test.Txt for Read
 If ReadStream.OpenFile("Test.Txt", 0) Then
            ‘File Test.Txt is Open for Read
            ALine = ReadStream.ReadLine
            ALine = ReadStream.ReadLine
            ReadStream.Close
End If
```

Issues

Remember the following regarding streams:

- Streams must be opened on the client.
- Streams may be either Read or Write, but not Read/Write
- Not all methods are supported by all stream types at this time.

Event Processing

Overview

One of the most powerful features in EsiObjects is event processing. Objects can watch other objects for events or changes in state. The TCP Bridge brings that same functionality to the client. Using the event processing capability of the bridge, a client can watch an object on the server for a specific event/state-change, or any event/state-change. When the event occurs, a specified method is invoked on the client (known as a "callback".) Thus clients can register for and respond to events that occur on the server.

One of the most common uses for event processing is for keeping a client display of data current with what is on the server. Especially when more than one process is changing data on the server. The client can register for events that occur when data is changed. When the event is thrown, the callback method is invoked and that method could update the client display with the most recent data.

In EsiObjects we say that an object will "watch" another object for a specific event or category of events. The watched object will "throw" the event when it encounters it by placing the event on an event queue.

Below we describe how a client makes use of event processing using the bridge.

Process description

There are several steps that must be done for event processing to be implemented on the client.

The programmer must define an EventSink. This is a class (.cls) file that contains the callback method for the event. It also manages a watch id that is a number assigned to each watch.

In the client code the programmer must:

1. Create an EventSink. This is an instance of the event sink class defined in the step above.
2. Create a "watch id" for the event sink which is a unique number assigned to the event and the sink.
3. Watch the object. This actually "registers" the fact that the client is going to watch a specified object for some type(s) of events. During the time that this watch is active, if the event occurs in the object, a specified callback method in the event sink is invoked.
4. When the client no longer wants to watch the object, it should ignore the object. This deactivates the Watch.

Each of these steps is detailed below.

Define an EventSink

Defining an EventSink involves creating a Class (.cls) file that contains the method(s) that will be invoked when an event occurs (the callbacks.) Below is an example of an event sink called MyEventSink that is defined in the MyEventSink.cls file. Only one callback routine has been defined. Note that the method is defined as Public. Also note that a public member is defined called WatchId. This is useful for storing the watch id with the sink.

A Generic Event Sink Example

```
File: MyEventSink.cls
‘Holder for the Watch Id
Public WatchId As Long
‘Generic Responder
Public Sub OnEvent(ParamArray Args() As Variant)
    ‘Args(0) = The Object that threw the event
    ‘Args(1) = The Event Name
    'Ok let's handle the Arg here
    'MsgBox ("Event on Object " + Args(0))
    Form1.List1.AddItem ("Event " + Args(1))
End Sub
```

Create an EventSink

To create an instance of the EventSink object, you use the new command. For example, if we want to create an event sink for the one defined in the example above:

```
Dim Sink As New MyEventSink
```

Create a WatchId based on an EventSink

Create a watch id that is associated with the EventSink. This watch id will be used to watch and ignore objects.

Watch an Object

Watching an object for an event is done by invoking the **Watch()** method on the Broker. This method call will specify the watch id and the EventSink that should handle the event when it occurs. In addition **Watch()** also specifies what method should be invoked on the EventSink when the event occurs.

Handle Events

The callback method in the EventSink implementation should handle the event as needed when it occurs.

Ignore Object

To stop watching an object for events the **Ignore()** method of the Broker is used. You can specify what event(s) to ignore or all events.

Free the WatchId

When there is no longer a reason to use the EventSink, the watch id associated with it should be released.

The Event Queue

Each client connection to the server is allocated an event queue. Whenever an event occurs the event is place on the event queue for the client to pick up. There are two ways in which the event queue has its events pulled and sent to the client:

At the conclusion of a call to the server, the event queue for the connection is checked and all of the events are dispatched back to the client at that time.

The **DispatchEvents()** method of the Broker may also be used to explicitly dispatch any events that might have been caused by the actions of others. This method essentially polls the server for any events on the queue and if any are found, they are dispatched to the client. Typically a Timer could be set up to execute this method at regular intervals to check for any events that have occurred.

The EventSink

The EventSink is an object (defined in a class file) that is called when an event occurs that the client is watching for. This object will be invoked at the specified method when the event occurs. The method that is run in response to the event (the callback) is specified on the **Watch()** method.

Event Signature Information (callback format)

The general signature for an event callback is

CallbackName(ParamArray Args() As Variant)

When an event is thrown, the specified callback in invoked and a variable number of arguments may be passed in from the server depending on the event. When watching for an event, you will always get at least two parameters:

1. The first parameter is the Object that generated the event.
2. The second parameter is the event name. The format of the name is “Interface::Name”.

Any other parameters that are passed are from the server implementation and depend on the event being thrown. To find out what parameters you can expect

from an event, check the documentation or implementation code for the event on the server.

Watching

Watching an object involves defining and creating an Event Sink, getting a watch id for the event, and then invoking the **Watch()** method. This method identifies the object you want to watch, the event(s) you are watching for, and the callback to be run when the event occurs.

GetWatchId

The **GetWatchId()** method on the Broker creates an identifier for a specific event sink. This watch id is used when creating and destroying watches to identify which object should receive the event. The Broker creates this id, and it is a good idea to store the watch id with the sink. (See the example code below.) The method takes one parameter. This parameter is the event sink object we are requesting a watch id for.

Example

```
WatchId=Broker.GetWatchId(Sink)
‘ store the watch id in the event sink
MyEventSink.WatchId=WatchId
```

Watch

The Broker implements a method called **Watch()** that allows a watch to be established on a specific object. The **Watch()** operation takes five arguments (see the reference section on the Broker below for more details).

1. The object to be watched
2. The event name or type of event to watch for
3. Watch Id
4. The event sink object
5. The callback method name in the event sink to be invoked when the event occurs.

Example

```
‘Setup a general Event Watch
Broker.Watch  theObject, "$Events", Sink.WatchId, Sink, "OnAnyEvent"
```

What can be watched

The second parameter to the Watch method specifies what is to be watched. A Watch may be established on an object for a specific event or category of event. Below is a list of the things that may be watched for by a client or another object.

Specific events – watch for a programmer-generated event.

Specific properties – watch for state changes to a property of an object.

Specific relationships – watch for changes/assignments to a relationship of an object.

Any Event – you can generically watch for all programmer-generated events.

Any Property – this will watch for state changes in any property or relationships.

Specifying what to Watch

Watching for a specific Event, Property, or Relationship

When a specific event, property or relationship is to be watched for, the format of the second parameter is simply a string containing the event, property or relationship name. If the item being watched for is in an interface other than Primary the format is “Interface::Item”.

Watching for any Property and Relationship change

The second parameter should be “$Properties” if you are watching for all property or relationship changes.

Watching for any Event

To watch for all programmer-generated events use “$Events” in the second parameter of **Watch()**.

Specifying who will handle the event

Arguments three, four and five in the **Watch()** method are used to determine where the event will be sent should it occur.

The Watch Id

Used as a master grouping token.

The Event Sink

The EventSink object that contains the callback method that is invoked to handle the event.

The Method

The name of the callback method in the sink that is called should the event occur.

Ignoring

Ignore

The **Ignore()** method is used to stop watching for events. Using this method all or some the watches made by under a watch id may be removed. See the reference section below for more details about the **Ignore()** operation.

Specifying what to Ignore

Depending on what arguments are passed to the **Ignore()** method various watches may be removed. A watch id is always required.

Ignoring a specific event, property, or relationship

The **Ignore()** method can be used to cancel a specific Watch that has been made on an object. The event, property, or relationship being ignored must have been explicitly watched via the **Watch()** method. When invoking the method this way, the object being ignored and the event name must be specified.

Ignoring all events

If all events for an object are being watched it is possible to remove this using the **Ignore()** method.

In invoking the method this way, the object being ignored and "$Events", to indicate all events, must be specified.

Ignoring all property or relationship watches

If all property changes for an object are being watched it is possible to remove this watch by specifying the object and the string "$Properties" on the **Ignore()**.

Ignoring all watches for an object

To remove ALL watches on a specific object the **Ignore()** is invoked with the object and an empty string ("") for the event name.

Turn off all watches for all objects

Using an empty string for both the object and event name on the **Ignore()** method will close out all watches will all objects on the server. You should use this in your application code before exiting your application to insure all client watches are cleaned up prior to closing.

Effect on Queued Items

When an **Ignore()** is issued, any events in the event queue corresponding to the events being ignored are filtered out automatically. If a **Watch()** is done during

this processing, any remaining items on the queue for the new watch will be dispatched normally..

FreeWatchId

When a watch id is no longer used this method will ensure that any resources associated with it will be freed. Any watches associated with this watch id will be removed.

Example of Event Setup

```
‘Create the EventSink Object
Dim Sink As New MyEventSink
‘Define the Watch id, and save it in the Sink
Sink.WatchId = Broker.GetWatchId(Sink)
‘Setup a general Event Watch
Broker.Watch  theObject, "$Events", Sink.WatchId, Sink, "OnAnyEvent"
‘Setup a specific Property watch for the Id Property
Broker.Watch theObject, “Id”, Sink.Watchid, Sink, “OnIdChange”
```

Example of Event Cleanup

```
‘Remove a general event watch
Broker.Ignore Sink.WatchId, theObject, “$Events”
‘Remove the watch of the property “Id”
Broker.Ignore Sink.WatchId, theObject, “Id”
‘Free the WatchId
Broker.FreeWatchId Sink.WatchId
‘Free the Sink
Set Sink = Nothing
```

Understanding When Events are Dispatched

There are two ways in which events are dispatched from the server event queue to the client:

At the conclusion of a call to the server (invoking a method, property, etc.), the event queue for the connection is checked and all of the events are dispatched back to the client at that time.

The **DispatchEvents()** method of the Broker may also be used to explicitly dispatch any events that might have been caused by the actions of others. This method essentially polls the server for any events on the queue and if any are found, they are dispatched to the client. Typically a Timer could be set up to execute this method at regular intervals to check for any events that have occurred. (See “Polling for events” below.)

Polling for Events

When your application is sitting idle, you may want to check the server to see if there are any events that have occurred. The Broker provides the method **DispatchEvents()** that will dispatch any events that may be enqueued on the server.

Using DispatchEvents

The usage of **DispatchEvents()** is rather simple. It simply needs to be invoked. It is suggested that this be done in either the applications idle time processing or by using a timer.

Example of Timer Code to Dispatch Events:

```
Private Sub Timer1_Timer()
    ‘Dispatch no more then 5 Events
    ESITCPBridge1.Broker.DispatchEvents 5
End Sub
```

Controlling Event Processing

There are several mechanisms for controlling event processing (all available in the Broker):

Event processing can be enabled/disabled using the **ChangeEventDispatch()** method.

The number of events outstanding can be checked using the **NumberOfEventsPresent()** method.

You can determine if event processing is enabled by using the **EventDispatchEnabled()** method.

Advanced Usage

Gateway Debugging Functions

The Bridge provides the services listed below for the purposes of executing EsiObjects commands or evaluating expressions within the context of an object on the server. These advanced services are useful for debugging purposes. Since code is being executed in the context of a server object, these should never be used on production code. Future deployment versions of the TCPBridge may not support these operations. Also, invoking these services may be subject to security restrictions.

ObjXecute

This service provides the ability to invoke arbitrary behavior in the context of an object. It uses EsiObjects syntax, similar to the Xecute command provided in EsiObjects or similar to the kind of commands that would be entered in the EsiObjects Xecute Shell. The value of $Return set during the executed code will be returned to the client

ObjEval

Provides the ability to invoke arbitrary behavior in the context of an object. You must supply a valid EsiObjects expression. The value of the expression will be returned to the client. This service is equivalent to ObjectXecute("Set $Ret="+Expr).

Xecute

Not yet implemented.

Using the Bridge with Proxies Disabled

Prior to Version 4.0 of EsiObjects, the ability to use proxies was not available. The only way to invoke behavior of an object was via an API on the Broker. This API, described here, allowed for invoking methods and set/get properties. Only positional parameter passing was allowed. This API has been retained for backward compatibility. Note that proxies return their Object Id as their default property, and thus can be used as the first argument to these calls.

Below we describe the API calls available. For details on the syntax of these services, see the reference section below.

InvokeMethod

Rather than using a proxy object it is possible to invoke a method on an object using the InvokeMethod method of the broker. Only positional parameter passing is supported. To invoke a method in an interface other than Primary, the format is “Interface::Method”.

PropertyGet

Rather than using a proxy object it is possible to get a property of an object using the PropertyGet method of the broker. Only positional parameter passing is supported. To invoke a property in an interface other than Primary, the format is “Interface::Property”.

PropertySet

Rather than using a proxy object it is possible to set a property of an object using the PropertySet method of the broker. Only positional parameter passing is supported. To invoke a property in an interface other than Primary, the format is “Interface::Property”.

Example

```
‘Lookup the object Id of the Environment
Env = Broker.LookupObject(“$ENV”)
‘Use to broker to get the Objects Name
Name = Broker.InvokeMethod(Env,”Name”,1)
‘Get the FullName Property Of the Environment
Fullname =Broker.PropertyGet(env,”FullName”)
‘Set the last error proprerty to Null
Broker.PropertySet Env, LastError, “NoError”
```

Unsupported Behavior

Property Accessors not currently supported

The following property accessors are not supported in the bridge (either by using proxies or the Broker API): $Data, $Get, $Order, $Query, $Normalize, $Valid, Kill

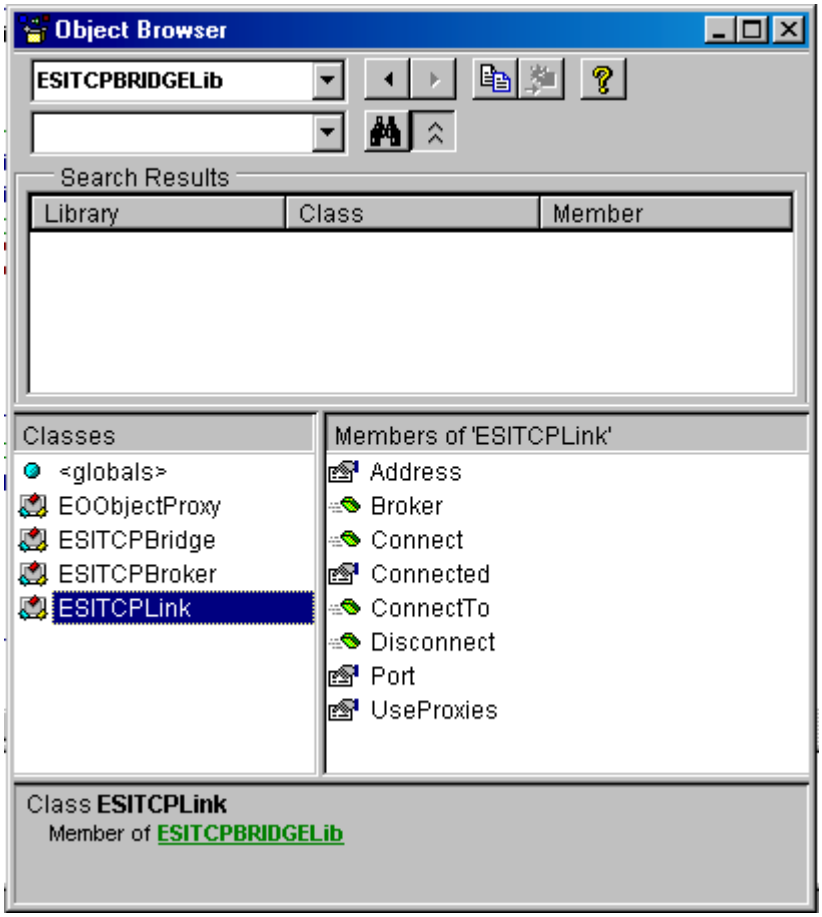
Out & In/Out parameter passing

Currently there is no mechanism for calling EsiObjects methods that use an Out, or a In/Out parameter pass mechanism. In the future the variant would need to be passed with a “by reference” flag set.

Access from ASP Pages

Using TCPLink

Provided with the Bridge is a TCPLink COM component that is similar to the TCPBridge but designed to work in a variety of clients. In specific the TCPLink may be used to communicate with EsiObjects from an ASP page on a Web Server.



The TCPLink component as shown in the VB Object Browser

Process

Creating the Link Object

Within an active server page, the link object is created by invoking the CreateObject service on the server.

```
Link=Server.CreateObject(“ESI.TCPLink”)
```

Connect to the Server

Connecting to a server via the TCPLink is similar to the connection mechanisms supported by TCP Bridge.

In other words, **Connect** or **ConnectTo** methods can be used. The properties **Address** and **Port** should be set accordingly when using **Connect()**.

Enable Proxies

Similar to the TCP Bridge if you wish to use Proxies, you should set the **UsesProxies** property to TRUE.

Using the Broker

With Proxies

Using Proxies with the TCPLink is similar to using Proxies as described above with the TCP Bridge.

To enable proxies, set the UseProxies to True.

```
Link.UseProxies = 1
```

Server objects created and returned to the client, once this property is set, are proxy objects that can be invoked like a proxy object on the TCP Bridge. The programmer requests a Broker object from the link by invoking the **Broker()** service.

```
Broker = Link.Broker()
```

The Broker object here is the same as the Broker object used by the TCP Bridge. This means you can use such services as **LookupObject()**, **CreateObject()**, etc. The use of proxy objects returned by the server is the same as well.

```
Set EnvObj = Broker.LookupObject("$ENV")
EnvObj.Assert "Hello World"
```

Without Proxies

Setting **UseProxies** property to False will disable the use of Proxies. The invocation of services on the server therefore, is via the API available in the Broker. Namely **InvokeMethod()**, **GetProperty()**, and **SetProperty()**.

```
Broker.InvokeMethod EnvObj, "Assert", 1 "Hello World"
```

Closing the Connection

At the end of the script the following actions should be done:

1. Clear the Broker and other objects (see below.)
2. Disconnect the link using the **Disconnect()** method.
3. Clear the link.

Clearing objects

Object references may be cleared by setting the object to **Nothing.** For example:

```
Set Object = Nothing.
```

Disconnecting the Link

Use the **Disconnect()** method of the link object to disconnect from the server.

Example

Here is the basic outline of and ASP script.

```
<%@ Language=VBScript %>
<%
      Set EOConnection=Server.CreateObject("ESI.TCPLink")
      EOConnection.Address = "appsrv4.esitechnology.com"
      EOConnection.Port = 9000
      EOConnection.UseProxies = 1

      temp = EOConnection.Connect()

      if temp <> True then
            Response.Write("<P>Connection Failed!</P>")
      end if

      Set Broker = EOConnection.Broker()
      Set Env = Broker.LookupObject("$ENV")

%>

<HTML>
<HEAD>

</HEAD>
<BODY>
<!--
       Body of Page goes here along with additional script calls
//-->
<P> The Environment Object is named
<%
     Response.Write Env.Name
%>
</P>
</BODY>
</HTML>
<%
      Set Env = Nothing
      Set Broker = Nothing
      EOConnection.Disconnect
      Set EOConnection = Nothing
%>
```

Reference

The TCPBridge

Methods

AboutBox

Displays the About Box for the ESITCPBridge,

Signature

Sub AboutBox()

Broker

Returns the Broker object for this connection.

Signature

Function Broker As ESITCPBroker

Connect

Connects to an EsiObjects server using the *Address* and *Port* properties to determine which server to connect to.

Signature

Function Connect() As Boolean

Returns: *True if the connection was made.*

ConnectTo

Connects to an EsiObjects server using the *Address* and *Port* input parameters to determine which server to connect to.

Signature

Function ConnectTo(Address As String, Port As Integer) As Boolean

Returns: *True if the connection was made.*

Parameters:

Address – the IP address of the server to connect to.

Port – the port at which the EsiObjects server is listening for connections.

Disconnect

Disconnects from the server.

Signature

Function Disconnect() As Boolean

Returns: *True if the bridge is disconnected from the server.*

Properties

Address

The IP address of the EsiObjects server to connect to.

Signature

Address As String

Examples

10.0.0.100

127.0.0.1 (loopback address)

appsrv4.esitechnology.com

AutoConnect

If this property is set to TRUE, the bridge will attempt to connect to the server when the control is loaded.

Signature

AutoConnect As Boolean

Connected

Boolean to indicate whether the bridge is connected to the server. TRUE means a connection exists.

Signature

Connected As Boolean

Port

The Port on the EsiObjects server is running on.

Signature

Port As Integer

Examples:

9000

2200

ProxyDefaultValue

If set to TRUE proxy obejects will have a default value of the OID of the object which they proxies.

Signature

ProxyDefaultValue As Boolean

ReturnNullString

If set to TRUE a null string returned from the server will create an empty variant. If set to FALSE then a NULL variant is created, which can be tested using the IsNull function in VBScript or Visual Basic.

Signature

ReturnNullString As Boolean

UsesProxies

If set to TRUE, then proxy objects will be created.

Signature

UsesProxies As Boolean

Events

Connect

Generated when a connection is made to the server

Signature

Event Connect()

Disconnect

Generated when the connection to the server is terminated

Signature

Event Disconnect()

OnBrowse

Generated when the server generates a symbol dump to the client. This will occur if the server encounters the ZVIEW command.

Signature

Event OnBrowse(Object As String, Title As String, Table As Object)

OnDiagnostic

Generated when a diagnostic message is generated on the server. For example, “Do $ENV.Trace(Msg)” would trigger this event.

Signature

Event OnDiagnostic(Text As String)

OnError

Generated when the server encounters an error. For example, Do $ENV.ReportError(“Message text”)

Signature

Event OnError(Message As String, Title As String)

OnWarning

Generated when the server issues a warning message. For example, Do $ENV.OnWarning(“Message text”)

Signature

Event OnWarning(Text As String)

Output

Generated when the Environment outputs text. For example, Do $ENV.Output(“Message text”)

Signature

Event Output(Text As String)

The Broker

Methods

ChangeEventDispatch

Sets the state of Event Processing by the Bridge.

Signature

Sub ChangeEventDispatch(Enabled As Long)

Parameters:

Enabled – set to 1 for enable, 0 to disable event processing.

ClearFault

Resets the fault state. If used in OnError the fault will not be passed on to the caller.

Signature

Sub ClearFault()

CreateObject

This method provides complete object creation service.

Signature

Function CreateObject(ClassName As String, Flags As Long, [ParameterList As EsiList], [NamedParameterList As EsiNvList], [KeywordList As EsiNvList], [PropertyList As EsiNvlist]) As Variant

Returns: The new object.

Parameters:

ClassName – the name of the class of the object to create.

Flags – creation flags.

1 – Shared

0 – Child

ParameterList – an EsiList of the positional parameters used for creation.

NamedParameterList – an EsiNvList of the named parameters to use in the creation.

KeyWordList – an EsiNvList of the Creation Keywords to use for the creation.

Keyword	Value
Base	The base location to use to create the object
Child	1 = Create as a Child object
Class	1 = Object should be created in the Class scope
Domain	A Domain to use to create the object
Fixed	A fixed location to use to create the object
Name	The name of the object in the Domain
Shared	1 = Create as a Shared Object

Keywords are case sensitive.

PropertyList – an EsiNvList of the property to assign during object creation.

DestroyObject

Destroys an object on the server.

Signature

Function DestroyObject(Object As String) As Boolean

Returns: True if successful.

Parameters:

Object – the object to destroy.

DispatchEvents

Polls the Server to see if there are any pending events and dispatches any that are found.

Signature

Sub DispatchEvents(Max As Long)

Parameters:

Max – the maximum number of events to dispatch. A value of zero (0) will dispatch all pending events.

EventDispatchEnabled

Determines if event dispatching is enabled.

Signature

Function EventDispatchEnabled() As Long

Returns: Zero(0) if Events are disabled.

FreeWatchId

Frees a watch id. All watches with this id will be removed.

Signature

Sub FreeWatchId(WatchId As Long)

Parameters:

WatchId – the watch id to free.

GetWatchId

Gets a watch id for a specific EventSink.

Signature

Function GetWatchId(EventSink As Object) As Long

Returns: The watch id.

Parameters:

EventSink –the Event Sink in which a watch id should be allocated.

Ignore

Removes watches from the server.

Signature

Sub Ignore(WatchId as Long, Object As String, Item As String)

Parameters:

WatchId – the watch id.

Object – the object to ignore. (A null string will ignore all objects.)

Item – the Event or Property to ignore. The format of the item should be "Interface::Item". The value "$Properties" or "$Events" may be used to ignore a

corresponding watch. A Null string will ignore all events & properties for this object.

InvokeMethod

Invoke a method on an object.

Signature

Function InvokeMethod(Object As String, Method As String, ParamArray VariableArguments() As Variant)

Returns: The return value of the method.

Parameters:

Object - the object receiving the method request.

Method – the name of the method to invoke. To invoke a method in an interface other than the Primary interface, the name must be in the format of "Interface::Method".

ParamArray – Variable length array of variants to be passed to the method as positional parameters.

LookupObject

The LookupObject is a mechanism for locating persistent and system objects on the server.

Signature

Function LookupObject(Name As String) As Variant

Returns: The value associated with the name or an Empty string.

Parameters:

Name – the name of the object to lookup. (See below.)

Notes on Names

Locating System Variables

The table below lists the system objects that the **LookupObject()** service may find. The names are not case sensitive.

System Object	Abbreviation	Description
$ENVIRONMENT	$ENV	The Environment object

		associated with this connections
$LIBRARY	$LIB	The default Library
$LIBRAYRLIST		The List of all Class Libraries
$SYSPOOL		The System Name Pool

Locating Class Objects

The **LookupObject()** service may be used to find the Class object associated with a class name. When looking up the class, the name used should be the full class name prefixed with an underscore. For example to find the class object the TimeStamp in the Base Class Library the name would be "_Base$TimeStamp".

The standard format for nested class names should also work, e.g. "_Lib$Class>Nest1>Nest2".

Locating an O% Name

LookupObject() may also be used to find named objects in the current default domain. When coding in EsiObjects such names are prefixed with an "O%". When using the **LookupObject()** service the O% should not be used, just the name. If a name is not found then an empty string will be returned. It is thus possible to check to see if name is defined by checking against the empty string.

NumberOfEventsPresent

Determines the number of events on the queue that need to be processed.

Signature

Function NumberOfEventPresent As Long

Returns: The number of event present in the Event Queue on the server.

ObjEval

Evaluates an expression in the context of an object on the server.

Signature

Function ObjEval(Object As String, Expr As String)

Returns: The value of the Expression.

Parameters:

Object – the object

Expr – the expression to evaluate.

ObjXecute

Executes code in the context of an object on the server.

Signature

Function ObjXecute(Object As String, Action As String)

Returns: The Value of $Return.

Parameter:

Object – the object.

Action – the code to execute in the context of the of the object

PropertyGet

Get the value of a property.

Signature

Function PropertyGet(Object As String, Property As String, ParamArray VariableArguments() As Variant)

Returns: The value of a property.

Parameters:

Object – the object.

Property – the property to invoke. To invoke a property in an interface other than the Primary interface, the name must be in the format of “Interface::Property”.

ParamArray – variable length array of variants to be passed to the property Value accessor as positional parameters.

PropertySet

Set the value of a property.

Signature

Function PropertySet(Object As String, Property As String, Value, ParamArray VariableArguments() As Variant)

Returns: True if the property was set.

Parameters:

Object – the object.

Property – the property to invoke. To invoke a property in an interface other than the Primary interface, the name must be in the format of “Interface::Property”.

Value – the new value of the property.

ParamArray – variable length array of variants to be passed to the property Assign accessor as positional parameters.

SimpleCreateObject

Simplified procedure to create an object on the server.

Signature

Function SimpleCreateObject(ClassName As String, Flags As Long, ParamArray VariableArguments() As Variant) As Variant

Returns: The new object.

Parameters:

ClassName – class name of the object to create.

Flags – creation flags.

1 – Shared

0 – Child

ParamArray – a variable list of parameters to use on as positional parameters in the Factory::CREATE method.

Watch

Establishes a watch on an object.

Signature

Sub Watch(Object As String, Item As String, WatchId As Long, Sink As Object, Method As String)

Parameters:

Object – the object to watch.

Item – the event or property to watch. The format of the name should be “Interface::Item”. The value “$Properties” may be used to watch all properties. The value “$Events” may be used to watch all events.

WatchId – the watch id.

Sink – the EventSink to invoke when the event occurs.

Method – the method to invoke on the sink when the event occurs.

Xecute

Executes code on the server. (Not yet implemented)

Signature

Function Xecute(Action As String)

Returns: The Value of $Return

Parameters:

Action – the EsiObjects code to execute.

The TCPLink

Overview

The TCPLink is a automation component that functions similar to the TCP Bridge without the extra facilities expected of an ActiveX control. The TCPLink is useful for clients that do not have a user interfaces, for example an ASP page.

Methods

Broker

Returns the Broker object for this connection.

Signature

Function Broker As ESITCPBroker

Connect

Connect to an EsiObjects server using the *Address* and *Port* properties to determine which server to connect to.

Signature

Function Connect() As Boolean

Returns: TRUE if the connection was successfully made.

ConnectTo

Signature

Function ConnectTo(Address As String, Port As Integer) As Boolean

Returns: TRUE if the connection was successfully made.

Parameters:

Address – the IP address of the server to connect to.

Port – the port at which the server is listening for connections.

Disconnect

Disconnects from the server.

Signature

Function Disconnect() As Boolean

Returns: True if the link is disconnected from the server.

Properties

Address

The IP address of the EsiObjects server to connect to.

Signature

Address As String

Examples

10.0.0.100

127.0.0.1

appsrv4.esitechnology.com

Connected

If TRUE, the link is connected to the server.

Signature

Connected As Boolean

Port

The port on which the EsiObjects server is running.

Signature

Port As Short

Examples:

9000

2200

ProxyDefaultValue

If set to TRUE proxy obejects will have a default value of the OID of the object which they proxies.

Signature

ProxyDefaultValue As Boolean

ReturnNullString

If set to TRUE a null string returned from the server will create an empty variant. If set to FALSE then a NULL variant is created, which can be tested using the IsNull function in VBScript or Visual Basic.

Signature

ReturnNullString As Boolean

UsesProxies

If set to TRUE, then proxy objects will be created.

Signature

UsesProxies As Boolean