Using the OPC Automation Wrapper

1. Introduction

1.1. Author's Note: The purpose of this document

The Automation Wrapper provides a relatively simple way for developers who have a reasonable grasp of Visual Basic or scripting languages such as Visual Basic for Applications or VBScript to create their own OPC clients.

In order to help understanding, I have provided guidance and example code on how to use the OPC Automation Wrapper as a **step-by-step tutorial**. It should be possible to follow through each of these steps, ending with a fully functioning (if simple) OPC client application.

During the course of this tutorial I will endeavour to provide guidelines and strategies for using the key functionality available within the wrapper library. At the time of writing I assume the majority of applications developed using the library will be implemented in Visual Basic and therefore all the examples presented are coded in Visual Basic 6. It should be relatively easy to adapt the code to other versions of Visual Basic or to VBA for use in Excel.

In brief, the objective of this tutorial is to:

- Provide a brief introduction to the OPC Automation Wrapper
- Provide a step-by-step demonstration of how to create an OPC Client application in Visual Basic that utilises the OPC Automation Wrapper

Dave Evans – Omron CX-Server OPC Senior Developer – August 2003

1.2 Who should read this document?

The target audience for this guide is application developers knowledgeable in industrial automation technology and Visual Basic or, Visual Basic for Applications (as used in Excel). No attempt is made to teach automation technology and terminology (e.g. SCADA) or Visual Basic.

Full knowledge of OPC is not required in order to read this guide. However, all OPC developers should have, and use, a separate OPC reference (e.g. as provided by the OPC foundation, see www.opcfoundation.org).

1.3 How should this document be used?

The information in this document is provided in the form of a tutorial. It is anticipated that many readers will choose to follow through the tutorial, creating their own OPC Automation Wrapper based client based on the examples given. However this is not essential - if preferred the document can simply be read.

All readers should be aware that the full source code for the final completed tutorial is installed with CX-Server OPC, and that considerable time can be saved by using this provided source code (e.g. copying and pasting it, particularly for the longer subroutines).

2. Introduction to the OPC Automation Wrapper

2.1 What is the OPC Automation Wrapper?

The OPC Automation Wrapper is an Automation (Visual Basic or script language) interface used to connect to OPC Servers. It is called a wrapper because it “wraps” the Custom (C++) programming interface, calls to the OPC Automation interface being automatically converted to calls to the OPC Custom interface.

Many people are aware of the OPC servers and clients created by the OPC Foundation members but few SCADA application developers are familiar with the OPC Automation Wrapper. There are also many toolkits provided by factory automation companies for the third party developer to use to create OPC clients and Servers. However, these toolkits often require the user to have a detailed knowledge of C++ and component-based (COM) programming. The OPC Automation Wrapper is intended to be simpler and easier to use, and is therefore ideally suited for medium sized OPC applications written in the Visual Basic programming language

In more detail, the automation wrapper is a library of programming (COM) objects that provides for browsing the OPC Server’s address space, creating groups and items and manipulating their functionality. The initial source code for the wrapper was written by the OPC Foundation, but members of the OPC Foundation (such as Omron) are permitted to modify and distribute the wrapper.

The COM objects that comprise the library are contained in the dynamic link library OmronDAAuto.dll and implement what is intended to be (and to a reasonable extent is) an intuitive model of the OPC Server. Detailed information about the methods and properties available on each of the objects is provided in the OPC Data Access Automation Interface standard v2.05. It is not the intention of this tutorial to detail each of the objects available, their methods and properties, but instead to give an insight into how they can be used to create Automation client applications.

Developers using CX-Server OPC should consider the Automation Wrapper as a central path way between the simplicity of the CX-Server OPC Communications Client component and the complexity and full power of the Custom interface.

This tutorial addresses that central path of development.

2.2 The relationship between the OPC Automation Wrapper and OPC Clients and Servers

The wrapper essentially translates between the language format required for an Automation client and the custom interface of the OPC Server.

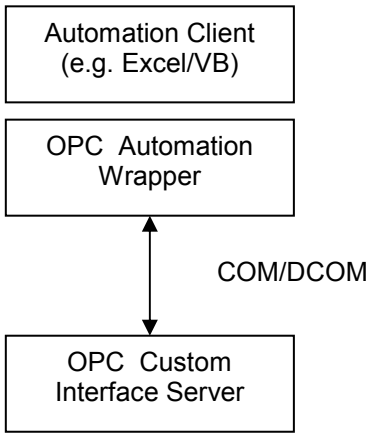

Figure 1 The Context of the Automation Wrapper

The Automation Wrapper allows for connection to OPC Servers as “in process” (i.e. DLL), local (i.e. to a .EXE running on the same machine) or remote (i.e. over a network, using DCOM) configurations.

2.3 The Automation Wrapper Object Model

The diagram below shows the object model for the Automation Wrapper. These objects provide an intuitive and object oriented approach to application creation. All applications using the Automation Wrapper should be designed with this model kept in mind.

The most important feature about the COM objects that comprise the library is that they reflect the components found within an OPC Server. The OPCServer is the root object from which access to the OPC Browser and groups can be obtained. The OPCGroups and OPCItems objects are collection classes and simply allow for the management of both Groups and Items - the functionality, methods and properties of both these classes are similar. The OPCBrowser object is used for inspecting the OPC server's address space.

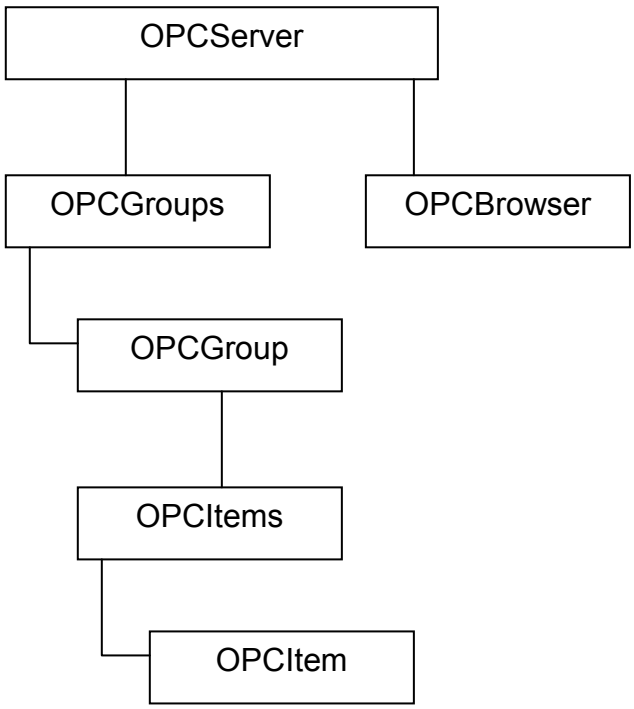

Figure 2 Automation Wrapper Object Model

Detailed information about each of the objects in the model can be obtained from the OPC Automation specification document.

3. Step-By-Step OPC Automation Wrapper Tutorial

The best way to understand the use of the OPC Automation Wrapper is to actually use it to create an OPC client application.

We'll create a general purpose Visual Basic application that can connect to any OPC Server. This tutorial application will allow the user to create named groups within the server and to browse the address space in order to subscribe to items that can be added to each group. The current value, quality and timestamp for each item in a group can then be displayed and updated at the requested rate for the associated group. Dialogs will allow the user to modify the attributes of both the groups and items.

Visual Basic is essentially a GUI oriented environment employing forms and controls for the display of data. In our example the data will be held within the server and represented by the objects that comprise the Automation Wrapper. Therefore the GUI elements will simply reflect the current state of the OPC objects.

In the previous section I touched briefly on the importance of understanding the Automation Object Model. In this example I hope to show you how those objects provide the central framework for OPC Automation Wrapper based applications and how they can be supported by dialogs and forms.

Note: The full source code for the final completed tutorial is installed with CX-Server OPC, and considerable time can be saved by using this provided source code (e.g. copying and pasting it, particularly for the longer subroutines). Also, please note that although this document includes all the **key** *code, the full source of the tutorial should be consulted to ensure a fully functioning application.*

3.1 Creating the Application

It is necessary to first create the application. To do this, open Visual Basic, select a new standard .EXE application and call it "**opcClient"**.

As this is intended this to be an application that uses the Automation Wrapper, we need to be able to utilise the OPC Data Access Automation library in our application - this is achieved by creating a reference to the library in Visual Basic. To do this first select the **Project/References** menu item and then add a reference to OMRON DA OPC Automation Library.

In this tutorial application we will also use the tree view and list view objects that are contained in the Microsoft Scripting Runtime Library, so add a references to:

- Microsoft Windows Common Controls 6.0 (SP4)
- Microsoft Forms 2.0

Now save the project in a suitable directory.

3.2 Creating the Main Form

The main window of the application will consist of a form with a tree view control showing each of the groups, and a list view showing each of the items configured for the currently selected group. A menu bar will provide a simple and easy method for invoking dialogs to manipulate each of the groups and items.

To achieve all of this, create a form with the name "**MainForm.frm**". Add a tree view on the left with the name "**treeGrps**". Add a list view on the right with the name "**listItems**". Set the "**View**" property of the list view to "**lvwReport**". Add to the list view the columns shown in the

figure below for "**Item**", "**Value**", "**Quality**" and "**TimeStamp**". Add a text box to the bottom of the form with the name "**txtMsg**".

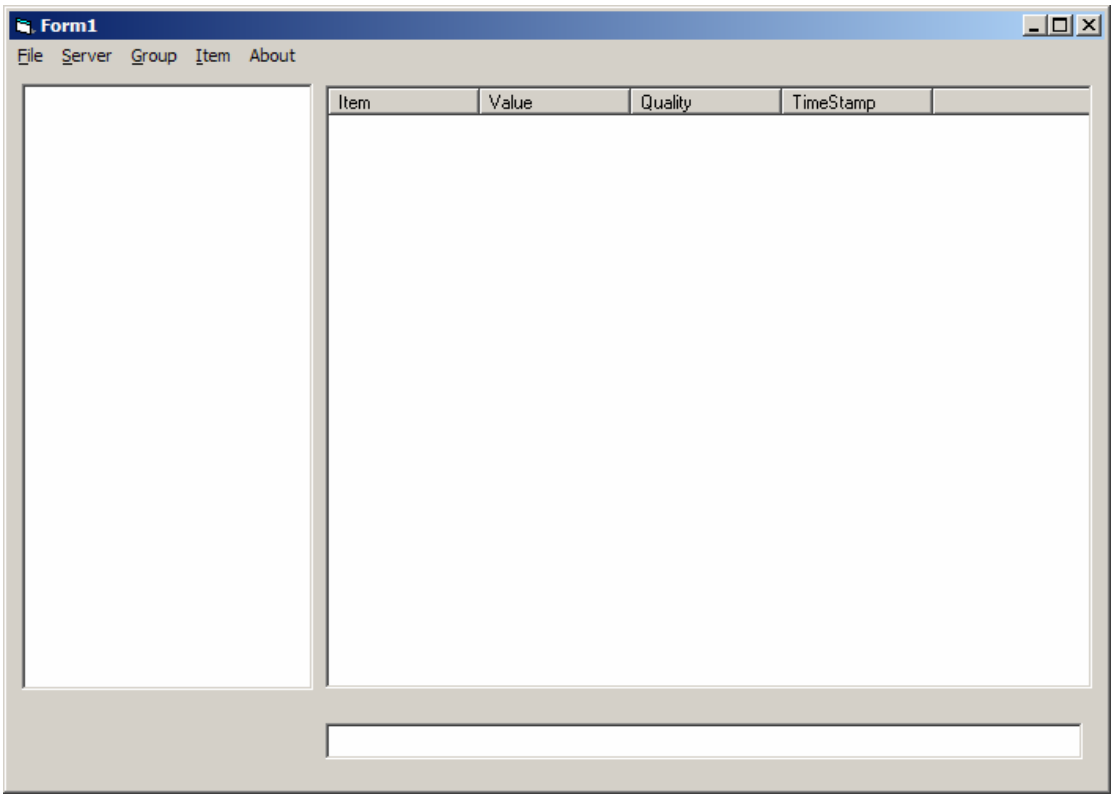

Figure 3 The main form

The groups created while using the application will be displayed in the tree view "**treeGrps**". Each group displayed will be associated with the icon of a folder that changes from closed to open when the node is selected on the tree. To do this it is necessary to associate an image list with the control. Create an image list control called "**ImageList1**". Add the bitmaps "**cGrp.bmp**" and "**oGrp.bmp**" (see Figure 4 - they can be found in the Tutorial source directory) using the names "**group**" and "**oGroup**" respectively.

Figure 4 Close & Open Group Icons

To associate the image list with the tree view add the following line to the **Form_Load()** subroutine of "**MainFrom.frm**":

```
    treeGrps.ImageList = ImageList1
```

Code: MainForm (MainForm.frm), Form_load

Now add a menu bar to the main form.

- Initialise the menu with the following top level menu items:
  - **File**,
  - **Server**
  - **Group**
  - **Item**
  - **About**

- Add the sub item "**Exit**" to the **File** menu.
- Add "**Connect**", "**Disconnect**" and "**Status**" to the **Server** menu.
- Add "**Add Group**", "**Remove Group**" and "**Edit Group**" to the **Group** menu.
- Add "**Add Item**", "**Remove Item**", "**Edit Item**", "**Async Read**" and "**Async Write**" to the **Item** menu.

Give each of the menu items an appropriate name. I usually use the hierarchy to form a name, e.g. I called the **Group|Delete** menu item "**mnuGroupRemove**".

The essential GUI elements are now in place for the main form and we can start to concentrate on linking them up with the objects that provide control over an OPC server.

3.3 Connecting to an OPC Server

Look back to the Automation Wrapper Object Model (see figure 2) and you'll see that at the root is the **OPCServer** object from which all the other objects are created and accessed. In this example I have created a global instance of the **OPCServer** object within a separate module "**Global.bas**". I've called this object "**theServer**". By doing this I can gain access to the object hierarchy easily from anywhere within the program. Later we'll add other appropriate global methods and variables to this module to enhance the application.

Connecting to a server, adding groups and items are carried out using a very simple approach via the menu on the main form. I could have used more sophisticated mechanisms using the tree view, but I wanted to keep the example simple and the amount of GUI code to a minimum in order to concentrate on illustrating how to use the OPC objects.

Under the "**Server**" menu there are three sub items to connect, disconnect and show the current status of the server. Before we can use an OPC server we need to locate one and connect to it. The methods for achieving this are provided by the **OPCServer** object. On selection of the **Connect** menu item we need to display a dialog that allows the user to select and connect to the required OPC server. To do this create a dialog called "**svrSelectDlg**" and add the GUI controls as shown below.

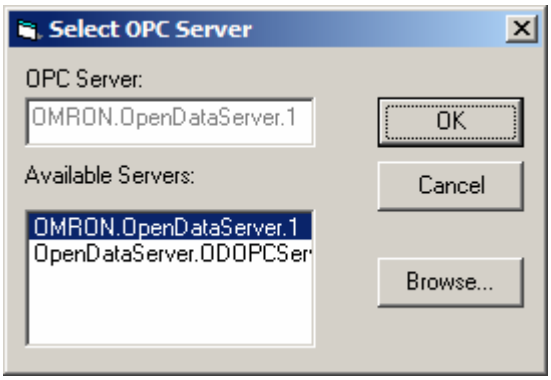

Figure 5 Server Selection Dialog

In the **Form_Load()** subroutine add the following code to initialise the dialog controls:

```
Private Sub Form_Load()
    Dim ServerNames As Variant
    Dim i As Integer
    Set theServer = New OPCServer
    ServerNames = theServer.GetOPCServers
    For i = LBound(ServerNames) To UBound(ServerNames)
        lstServers.AddItem ServerNames(i)
    Next
    lstServers.ListIndex = 0
```

```
        txtServer.Text = lstServers.List(0)
    End Sub
```

Code: SvrSelectDlg (svrSelectFrm.frm), Form_Load

Notice how I setup the Global variable "**theServer**" with a new **OPCServer** object and then request the names of the available OPC servers. I then iterate through the list and populate the list box.

For ease of use, in this application I only search the local PC for available servers. I do this by using the **GetOPCServers** method to provide a list of the locally available OPC servers. If you want to list the servers on a remote PC then you can pass the ID of the required PC to the **OPCServer** object using the same routine. I've put an extra button on the dialog so that you can add a browse facility and later you could adapt the code as necessary to search remote computers for OPC servers.

Having selected the identifier of the server you need to connect to it. The OPCServer object provides a subroutine for doing this called "**Connect**" that takes as its arguments the identifier of the server, and optionally the machine name if you want to attach to a remote computer (the default is to the local PC).

We need to enter the code that should be activated when the OK button is selected:

```
    Private Sub cmd_OK_Click()
        theServer.Connect (lstServers.List(lstServers.ListIndex))
        Unload Me
    End Sub
```

Code: SvrSelectDlg (svrSelectFrm.frm), cmd_OK_Click

If the Cancel button is selected we need to destroy the OPCServer object created at the start. This can be done as follows:

```
    Private Sub cmdCancel_Click()
        Set theServer = Nothing
        Unload Me
    End Sub
```

Code: SvrSelectDlg (svrSelectFrm.frm), cmdCancel_Click

Finally we need to add a method that responds to the click event on the list box in order to identify the selected server.

```
    Private Sub lstServers_Click()
        txtServer.Text = lstServers.List(lstServers.ListIndex)
    End Sub
```

Code: SvrSelectDlg (svrSelectFrm.frm), lstServers_Click

3.4 Disconnecting From the OPC Server

To disconnect from a server call the method **Disconnect** on the OPCServer object. If no other clients are attached to the server this will cause it to shut down immediately. Add this call directly into the method created to respond to the **Server** menu **Disconnect** command, i.e. the **mnuServerDisconnect** sub item in the main form:

```
      Private Sub mnuServerDisconnect_Click()
          Disconnect
      End Sub
```

Code: MainForm (MainFrm.frm), mnuServerDisconnect_Click

3.5 Getting the Server Status

Once an OPCServer object has been created and a connection made to a server, the status of the server can be determined. The **Server** menu has a **Status** sub menu item that should display a dialog that shows the current status of the server, as seen below.

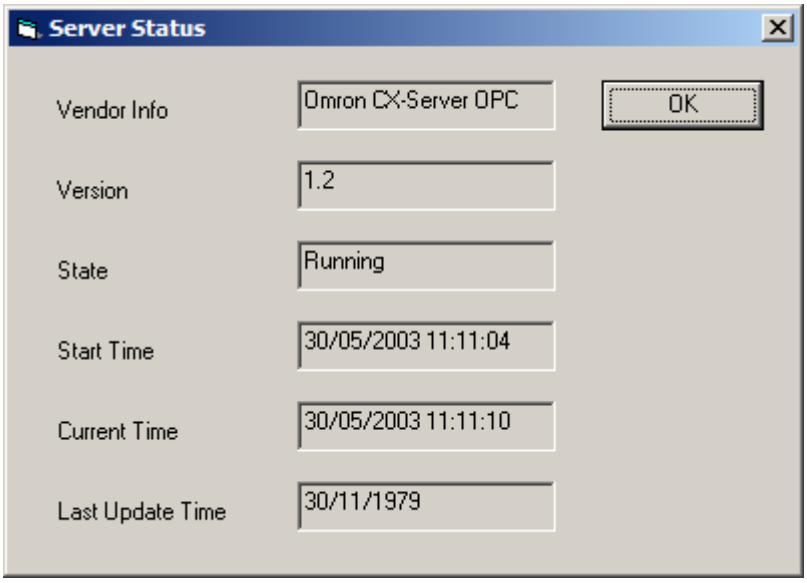

Figure 6 The Server Status Dialog

To achieve this functionality, create a dialog called "**StatusDlg**" and add the controls shown above. Invoke the dialog from the **Status item** on the **Server** menu.

In the **Form_Load** method for the dialog add the following code to access OPCServer object properties.

```
      Private Sub Form_Load()
          Dim theStates As Variant
          theStates = Array("Disconnected", "Running", "Failed", "No
Configuration", _
           "Suspended", "In Test")
          txtVendor = theServer.VendorInfo
          txtVersion = theServer.MajorVersion & "." &
theServer.MinorVersion
          txtState = theStates(theServer.ServerState)
          txtStartTime = theServer.StartTime
          txtCurrent = theServer.CurrentTime
          txtLast = theServer.LastUpdateTime
      End Sub
```

Code: StatusDlg (StatusDlg.frm), Form_Load

3.6 Creating Groups

Once you've connected to a server you can create groups. In the main form we should set up a procedure that enables or disables the menu selections depending on whether a server has been connected and groups and items have been created. This protects against the user trying to carry out an operation for which the associated object does not exist.

Here is an implementation of the EnableMenuItems subroutine, which can be added in the main form (Note that it is not necessary to add *all* of this code at this point, but that it will be referenced later in this tutorial.):

```
Private Sub EnableMenuItems()
    If theServer Is Nothing Then
        mnuGroupAdd.Enabled = False
        mnuServerStatus.Enabled = False
        mnuServerDisconnect.Enabled = False
        mnuServerConnect.Enabled = True
    Else
        mnuGroupAdd.Enabled = True
        mnuServerStatus.Enabled = True
        mnuServerDisconnect.Enabled = True
        mnuServerConnect.Enabled = False
    End If

    If selectedGroup Is Nothing Then
        mnuItemAdd.Enabled = False
        mnuGroupRemove.Enabled = False
        mnuGroupEdit.Enabled = False
    Else
        mnuItemAdd.Enabled = True
        mnuGroupRemove.Enabled = True
        mnuGroupEdit.Enabled = True
    End If

    If selectedItem Is Nothing Then
        mnuItemRemove.Enabled = False
        mnuItemEdit.Enabled = False
        mnuItemAsyncRead.Enabled = False
        mnuItemAsyncWrite.Enabled = False
    Else
        mnuItemRemove.Enabled = True
        mnuItemEdit.Enabled = True
        mnuItemAsyncRead.Enabled = True
        mnuItemAsyncWrite.Enabled = True
    End If
End Sub
```

Code: MainForm (MainForm.frm), EnableMenuItems

To create a group you first need to gather together the information about the group. This can be done using the GroupDlg shown below which will be invoked when the **Add Group** menu item is selected.

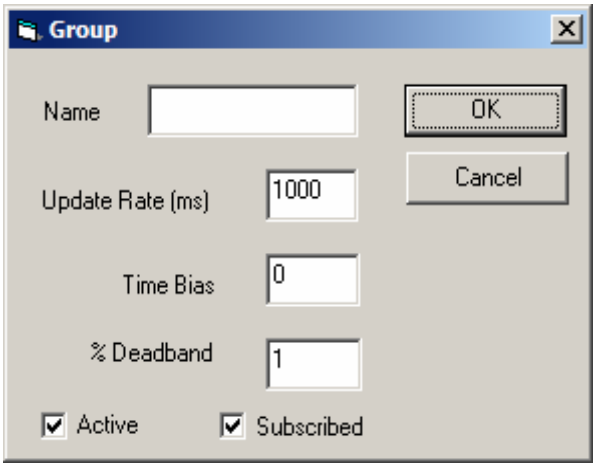

Figure 7 GroupDlg Dialog for creating a group

To add this functionality, create a dialog called GroupDlg with the controls shown in figure 7 and initialise them with the values shown. The intended default properties for a new group are initialised by default in the dialog and set when the OK button is selected. The server should create a default name for the group if one has not been assigned so it is acceptable to leave the Name edit box blank.

The Group dialog has two purposes; to create new groups in the server and to edit the properties of an existing group. In order to achieve this dual functionality the dialog must be initialised in one of two ways depending on the mode of operation. To achieve this declare a public variable in the dialog code:

```
Public theGroup As OPCGroup
```

Code: GroupDlg (GroupDlg.frm)

This public variable acts as a flag and a variable reference for a new or existing group. If a group is currently selected then before showing this dialog **theGroup** must be initialised using the currently selected groups variable, and the controls on the form will be initialised with the group's properties, otherwise the controls will retain their default values. Use of the selected group variable, which can be declared within the main form and defined as:

```
Public WithEvents selectedGroup As OPCGroup
```

Code: MainForm (MainForm.frm)

will be discussed later in a later section. Now add the following code to the Form_Load() subroutine.

```
Private Sub Form_Load()
    If Not theGroup Is Nothing Then
        txtName = theGroup.name
        txtRate = theGroup.UpdateRate
        txtDeadband = theGroup.DeadBand
        txtTimeBias = theGroup.TimeBias
        chkActive.Value = CInt(theGroup.IsActive) * -1
        chkSubscribe.Value = CInt(theGroup.IsSubscribed) * -1
    End If
End Sub
```

Code: GroupDlg (GroupDlg.frm), Form_Load

Notice how I test the value of theGroup variable and use a small trick to initialise the check boxes. In Visual Basic FALSE is represented by a value of FFFF or –1. A check box can have three state values 0, 1 or 2. We only require the unchecked or checked state, so by coercing the Boolean value to an integer value and multiplying by –1 we can obtain either the 0 or 1 state.

This application only connects to one server at a time and we're only interested in managing groups associated with that particular server, so add a global variable called theGroups in the Global module that will allow access to the groups collection for that server. This is really just for ease of use since it can also be obtained from theServer object each time it's required. Remember that when you use a COM object like this you need to use the **Set** directive otherwise you'll get a syntax error. The groups collection, theGroups is defined in the global.bas file as

```
Public theGroups As OPCGroups
```

Code: Global (Global.bas)

The group is added to the groups collection when the **OK** button is selected. This can be done using the code shown below. Notice how I set the variable **theGroup** to the new object created in the collection and then set the properties for the group from the entries in the dialog. Notice also on the dialog above (figure 7) that I set the dialog defaults for the active and subscribed properties to true.

```
Private Sub OKButton_Click()
    If theGroups Is Nothing Then
        Set theGroups = theServer.OPCGroups
    End If
    If theGroup Is Nothing Then
        Set theGroup = theGroups.Add(txtName.Text)
        txtName = theGroup.name
    End If
    theGroup.name = txtName
    theGroup.UpdateRate = CLng(txtRate)
    theGroup.DeadBand = CLng(txtDeadband)
    theGroup.TimeBias = CLng(txtTimeBias)
    theGroup.IsActive = CBool(chkActive.Value)
    theGroup.IsSubscribed = CBool(chkSubscribe.Value)
    Unload Me
End Sub
```

Code: GroupDlg (GroupDlg.frm), OKButton_Click

If the group is added then txtName needs to be reassigned because the server will automatically assign a name to the group if none is specified, and this is the name that should be used.

When returning from the Group dialog to the main form we now need to update the tree view that displays a list of all the Groups. To do this you can use the code in the subroutine **RefreshGroupTree**() shown below. It is not the purpose of this tutorial to explain the use of windows controls, so I'm not going to discuss how to use a tree view control in any detail. Instead we will examine the use of the **OPCGroup** and **OPCGroups** objects.

To refresh the view I simply clear it and recreate it, although if you had a very large tree view with a lot of data to be reconstructed this method would be very inefficient. For our purposes it

will suffice. Notice how **theGroups.Count** property is accessed and the assignment to **theGroup** variable, also that **theGroups** collection of items starts at an index of 1. (The variable **selectedGroup** is explained in the next section.)

```
      Private Sub RefreshGroupTree()
          Dim i As Integer
          Dim nd As Node
          Dim theGroup As OPCGroup
          treeGrps.Nodes.Clear
          If Not theGroups Is Nothing Then
              For i = 1 To theGroups.Count
                  Set theGroup = theGroups.Item(i)
                  Set nd = treeGrps.Nodes.Add(, , , theGroup.name,
"Group", "oGroup")
                  If Not selectedGroup Is Nothing Then
                      If nd.Text = selectedGroup.name Then
                          nd.Selected = True
                      End If
                  End If
              Next
          End If
      End Sub
```

Code: MainForm (MainForm.frm), RefreshGroupTree

Now we have a group available, we can add some items. This is where things start to hot up a bit, as we need to be able to interrogate the server to find out what's available.

In the next section we'll look at using the **OPCBrowser** object that allows access to the server's address space.

3.7 Browsing the Server

Once a group has been created it can be selected and items added. As mentioned earlier, I declared a variable within the main form called **selectedGroup**, which is defined as

```
      Public WithEvents selectedGroup As OPCGroup
```

Code: MainForm (MainForm.frm)

This statement indicates that the form will receive events from the COM object **selectedGroup** which is an OPCGroup object. When the user selects a group in the Group tree view the **selectedGroup** variable is assigned to the corresponding group. The significance of the WithEvents keyword will be discussed later.

When a group has been selected the **Add Items** menu is enabled, so that users can now choose to add an item. When they do, the Dialog "**BrowseDlg**" shown below is displayed.

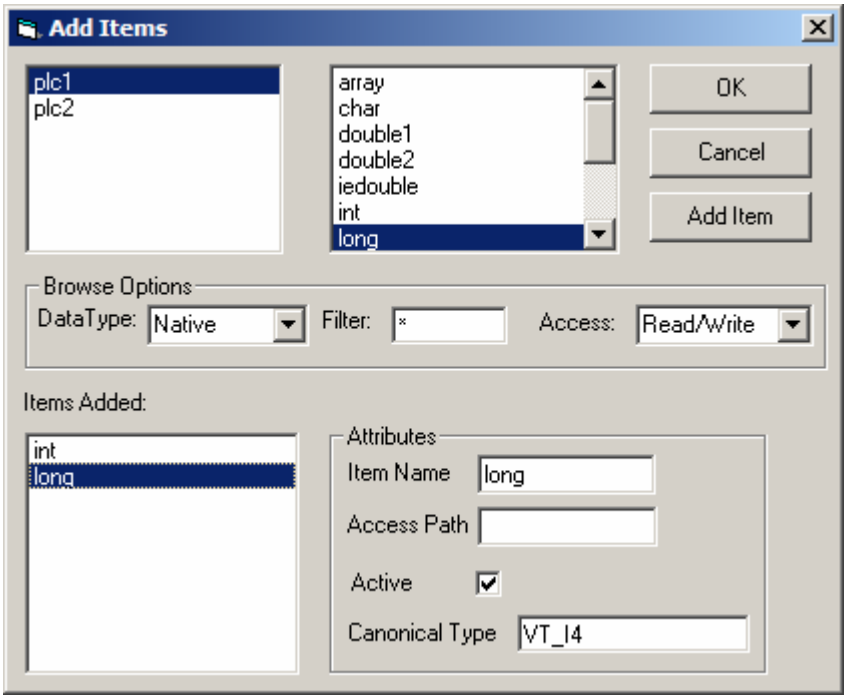

Figure 8 Dialog for Browsing Server and Adding items to Selected Group

This dialog allows the user to browse the server's address space and add items to the selected group. There are many ways in which I could have structured the code to achieve this functionality, but throughout this example I have purposely tried to keep the code to a minimum. I have therefore not created intermediate lists of items and properties associated with the items that the user wishes to add.

OK, now on to the business of browsing the server. To do this, create a dialog form called "**BrowseDlg**" similar to that depicted above and add the controls shown. The list box in the top left should be called "**lstBranch**", the list box in the top centre should be called "**lstLeaf**" and the list box in the bottom left should be called "**lstAdd**".

If you look back at the Automation Wrapper Object Model you'll see that the **OPCBrowser** object is associated with the **OPCServer** object, so to use the **OPCBrowser** we need to ask the **OPCServer** object to create one. This can be done using the form load subroutine seen below:

```
Private Sub Form_Load()
    Set theBrowser = theServer.CreateBrowser
    theBrowser.MoveToRoot
    theBrowser.ShowBranches
    PopulateBranch
    lstBranch.ListIndex = 0
    InitTypeCombo
    InitAccessCombo
    PopulateLeaf (lstBranch.List(0))
End Sub
```

Code: BrowseDlg (BrowseDlg.frm), Form_Load

The combo lists in the centre of the dialog control the filtering and are initialised using the subroutines below:

```
Private Sub InitTypeCombo()
    cmbBrowseType.Clear
```

```
        cmbBrowseType.AddItem "Native"
        cmbBrowseType.ItemData(cmbBrowseType.ListCount-1) = vbEmpty
        cmbBrowseType.AddItem "Short"
        cmbBrowseType.ItemData(cmbBrowseType.ListCount-1)= bInteger
        cmbBrowseType.AddItem "Long"
        cmbBrowseType.ItemData(cmbBrowseType.ListCount- 1) = vbLong
        cmbBrowseType.AddItem "Float"
        cmbBrowseType.ItemData(cmbBrowseType.ListCount-1)= vbSingle
        cmbBrowseType.AddItem "Double"
        cmbBrowseType.ItemData(cmbBrowseType.ListCount-1)= vbDouble
        cmbBrowseType.AddItem "String"
        cmbBrowseType.ItemData(cmbBrowseType.ListCount-1)= vbString
        cmbBrowseType.AddItem "Boolean"
        cmbBrowseType.ItemData(cmbBrowseType.ListCount-1)=vbBoolean
        cmbBrowseType.AddItem "Byte"
        cmbBrowseType.ItemData(cmbBrowseType.ListCount-1)= vbByte
        cmbBrowseType.ListIndex = 0
    End Sub
```

Code: BrowseDlg (BrowseDlg.frm), InitTypeCombo

```
    Private Sub InitAccessCombo()
        cmbBrowseAccess.Clear
        cmbBrowseAccess.AddItem "Read/Write"
        cmbBrowseAccess.ItemData(cmbBrowseAccess.ListCount - 1) =
OPCReadable Or OPCWritable
        cmbBrowseAccess.AddItem "Read"
        cmbBrowseAccess.ItemData(cmbBrowseAccess.ListCount - 1) =
OPCReadable
        cmbBrowseAccess.AddItem "Write"
        cmbBrowseAccess.ItemData(cmbBrowseAccess.ListCount - 1) =
OPCWritable
        cmbBrowseAccess.ListIndex = 0
    End Sub
```

Code: BrowseDlg (BrowseDlg.frm), InitTypeCombo

As you can see from the picture of the browse dialog above there are two list views that display the branches and leaves contained in the server address space. Most server address spaces are divided up into branches and leaves, rather than flat. The following subroutines can be used to populate the list boxes:

```
    Private Sub PopulateBranch()
        Dim i As Integer
        Dim name As String
        For i = 1 To theBrowser.Count
            name = theBrowser.Item(i)
            lstBranch.AddItem (name)
        Next
    End Sub
```

Code: BrowseDlg (BrowseDlg.frm), PopulateBranch

```
    Private Sub PopulateLeaf(branch As String)
    Dim i As Integer
    Dim name As String
        lstLeaf.Clear
```

```
        On Error GoTo EndRoutine
        theBrowser.MoveDown (branch)
        theBrowser.ShowLeafs
        For i = 1 To theBrowser.Count
            name = theBrowser.Item(i)
            lstLeaf.AddItem (name)
        Next
    EndRoutine:
        theBrowser.MoveUp
    End Sub
```

Code: BrowseDlg (BrowseDlg.frm), PopulateLeaf

The **OPCBrowser** object allows the programmer to effectively position at a particular branch and generate a collection of the leaves at that branch according to the current filter criteria. The filter criteria is controlled by setting the **OPCBrowser** attributes for type, access rights and name filter.

```
    Private Sub txtFilter_Change()
        theBrowser.Filter = txtFilter.Text
        PopulateLeaf (lstBranch.List(lstBranch.ListIndex))
    End Sub
```

Code: BrowseDlg (BrowseDlg.frm), txtFilter_Change

This method changes the items displayed in the leaf list box according to the filter property applied to item names.

3.8 Adding an Item to a Group

Having selected an item from the server address space we now need to add it to the currently selected group. In the main form prior to calling the BrowseDlg dialog I set the global variable theGroup to the selectedGroup. This OPCGroup variable can now be used to create a new item when the "**Add** *Item*" button is invoked:

```
    Private Sub cmdAdd_Click()
        Dim theItem As OPCItem
        Dim name As String
        name = lstLeaf.List(lstLeaf.ListIndex)
        lstAdd.AddItem (name)
        Set theItem = theGroup.OPCItems.AddItem(name,
currentHandle)
        lstAdd.ItemData(lstAdd.ListCount - 1) =
theItem.ServerHandle
        currentHandle = currentHandle + 1
    End Sub
```

Code: BrowseDlg (BrowseDlg.frm), cmdAdd_Click

In this subroutine I have declared a reference to an **OPCItem** object called **theItem.** I assign this when the new OPC item is created in the **OPCItems** collection belonging to the current group.

During item creation the new OPC item is assigned a client handle. I have used a global incremental identifier here for simplicity, which needs to be declared in "**global.bas**". In more complex applications it may be necessary to use a more sophisticated approach that assigns a client handle allocated from a dictionary.

Notice how I assign the **ServerHandle** of the new item to the **ItemData** array of the **lstAdd** list box that contains a list of all the items added. This is stored so that the item can be retrieved from the group using the handle when the name of the item is selected in the lstAdd list box, or if the operation is cancelled and the items need to be deleted.

3.9 Displaying Item Data in the List View

Once an item has been added to a group and the **BrowseDlg** dialog closed the item list on the main form needs to be updated. On returning from the dialog the method **RefreshItemList** shown below can be called to create the item list view:

```
      Private Sub RefreshItemList()
          Dim i As Long
          Dim li As ListItem
          Dim liSub As ListSubItem
          Dim theItem As OPCItem
          Dim v As Variant
          listItems.listItems.Clear
          If Not selectedGroup Is Nothing Then
              For i = 1 To selectedGroup.OPCItems.Count
                  Set theItem = selectedGroup.OPCItems.Item(i)
                  Set li = listItems.listItems.Add(i, ,
theItem.ItemID)
                  v = theItem.Value
                  Set liSub = li.ListSubItems.Add(, , v)
                  Set liSub = li.ListSubItems.Add(, ,
GetQualityText(theItem.quality))
                  Set liSub = li.ListSubItems.Add(, ,
theItem.TimeStamp)
                  li.Tag = CStr(theItem.ClientHandle)
              Next
          End If
      End Sub
```

Code: MainForm (MainForm.frm), RefreshItemList

This method iterates through the items of the selected group and builds the associated entry in the list view. Notice of the assignment of the client handle to the tag property of a list item. This will be used later for referencing the associated **OPCItem**.

The method **GetQualityText** called in the above code is defined below:

```
      Private Function GetQualityText(quality As Long) As String
          Select Case quality
              Case &HC0
                  GetQualityText = "Good"
              Case 1
                  GetQualityText = "Uncertain"
              Case 0
                  GetQualityText = "Bad"
          End Select
      End Function
```

Code: MainForm (MainForm.frm), GetQualityText

3.10 Modifying Item Properties

The **OPCItem** object has a number of properties associated with it including the current value, active state etc. Selection of an item in the **lstAdd** list box allows the active state of the item to be modified. This defines whether the value, quality and timestamp associated with the item are passed back to the **OPCGroup** object by the server if the **OPCGroup** object is active and subscribed. When an item is selected from the **lstAdd** list box the server handle is extracted from the **lstAdd**.ItemData array and used to access the associated **OPCItem**. Information about the selected item is displayed in the attributes area.

In order to determine the canonical (data) type of the item I have created a global method called **GetCanonicalType** in "**global.bas**" that creates a string representation of the type in terms of Variant types. (See the example code for when this is called):

```
      Public Function GetCanonicalType(theType As Integer) As String
          Dim Types As Variant
      Types = Array("VT_EMPTY", "VT_NULL", "VT_I2", "VT_I4", "VT_R4",
"VT_R8", "VT_CY", "VT_DATE", "VT_BSTR", "VT_DISPATCH", "VT_ERROR",
"VT_BOOL", "VT_VARIANT", "VT_UNKNOWN", "VT_DECIMAL",
            "VT_SPARE", "VT_I1", "VT_UI1", "VT_UI2", "VT_UI4",
"VT_I8")
          If theType > vbArray Then ' it's an array
              GetCanonicalType = "Array of " & Types(theType -
vbArray)
          Else
              GetCanonicalType = Types(theType)
          End If
      End Function
```

Code: Global (Global.bas), GetCannonicalType

3.11 Removing Items

When the **cancel** button is pressed it is necessary to remove items from the group. As with other access to the group an item's server handle is required to uniquely identify the item. The code below removes all the items currently held in the **lstAdd** listbox:

```
      Private Sub CancelButton_Click()
          Set theBrowser = Nothing
          Dim i As Integer
          Dim handles() As Long
          Dim Errors() As Long
          'need to remove any items added to group
          'here if there are any
          If lstAdd.ListCount > 0 Then
              ReDim handles(lstAdd.ListCount)
              For i = 1 To lstAdd.ListCount
                  handles(i) = CLng(lstAdd.ItemData(i - 1))
              Next
              theGroup.OPCItems.Remove lstAdd.ListCount, handles,
Errors
          End If
          Unload Me
      End Sub
```

Code: BrowseDlg (BrowseDlg.frm), CancelButton_Click

Notice how the server handles are added to an array and then removed in a single call to the server. I *could* also have created this subroutine in the following way, removing each item separately, but although simpler this is much less efficient.

```
      Private Sub CancelButton_Click()
          'less efficient example of removing items. See alternative
above
          Set theBrowser = Nothing
          Dim i As Integer
          Dim handle(1) As Long
          'need to remove any items added to group here if there are
any
          For i = 1 To lstAdd.ListCount
              handle(1) = CLng(lstAdd.ItemData(i - 1))
              theGroup.OPCItems.Remove 1, handle, Errors
          Next
          Unload Me
      End Sub
```

Code: BrowseDlg (BrowseDlg.frm), CancelButton_Click

In the same way that it's possible to remove multiple items from a group, it is also possible to add multiple items using the **AddItems** method on the **OPCItems** collection. This is particularly useful if your application needs to add a large number of items, e.g. when the initialisation information the groups and items to be created is stored in a file or database.

3.12 Updating Item Value Data

In an earlier section I mentioned the use of WithEvents in the context of the declaration of the variable **selectedGroup.** This statement creates a connection point to the **OPCGroup** object and associates a method which will be called when a specific event is raised.

This event can be handled using the following code which should be placed in the in the main form:

```
      Private Sub selectedGroup_DataChange(ByVal TransactionID As
Long, ByVal NumItems As Long, ClientHandles() As Long, ItemValues()
As Variant, Qualities() As Long, TimeStamps() As Date)
          Dim li As ListItem
          Dim i As Long
          For i = 1 To NumItems
              Set li = listItems.FindItem(CStr(ClientHandles(i)),
lvwTag)
              If Not li Is Nothing Then
                  li.ListSubItems(1).Text = ItemValues(i)
                  li.ListSubItems(2).Text =
GetQualityText(Qualities(i))
                  li.ListSubItems(3).Text = TimeStamps(i)
              End If
          Next
      End Sub
```

Code: MainForm (MainForm.frm), selectedGroup_DataChange

This method responds to the **DataChange** event and passes in arrays of client handles, values, quality identifiers and time stamps for each item that has changed during the requested time period. If the item is not active or has not changed then no update will be received for that item. In addition if the group is not active or not subscribed then no updates for any of the items in the group will be received.

The approach I have taken to displaying the updated values of items is to iterate through the array of client handles, search the tag field in the **listItems** for the corresponding entry and set the sub items accordingly.

A simpler approach would have been to call the subroutine **RefreshItemList** that would have rebuilt the whole list, but in applications where there are a large number of items in the group this latter approach would be very inefficient. In applications where the group size remains fixed or where an intermediate collection is used to manage the items the client handle for an item can correspond to an index or key for faster access. Developers must use experience and judgement to decide on the correct implementation to satisfy the requirements of their own applications.

The use of the **DataChange** event is important. In my client I only pick up the event on the currently selected group because this is the only group on display. In more complex applications it may be necessary to process the events on a number of groups simultaneously. In such situations the events for each individual group can be responded to, or the **GlobalDataChange** event that is fired from the **OPCGroups** object can be used. It is a function of the automation client that both the **DataChange** and **GlobalDataChange** events are fired simultaneously.

It should be noted, however, that use of the latter may cause bottlenecks in large applications because all the returned data for all the groups defined will come back to the client through the one event. It's best to choose one or the other method depending on your application but unwise to use both, as it will make your application unnecessarily complex and it can end up processing events with duplicate data.

We are now at the stage where we have a basic client application that allows us to make a connection to a local server, create groups and items and monitor the item values on selection of the group.

In the next section we'll look at the editing of items. We'll also look at synchronous and asynchronous methods for reading and writing data.

3.13 Editing Items

During the development of this tutorial I have tried to keep the design and implementation as simple as possible, using as the base for the application the object model implemented by the Automation Wrapper. In this part I'm going to add to the model features for editing items and reading and writing their values.

To edit an item's value we need to develop a new dialog that displays the item's value.

As with the group dialog first create a public variable "**theItem**". This will be set to the selected item in the main form prior to the item dialog being invoked.

```
Public theItem As OPCItem
```

Code: ItemDlg (ItemDlg.frm)

Then create a dialog form similar to that shown below called "**ItemDlg**" and initialise the controls in the **Form_Load()** subroutine.

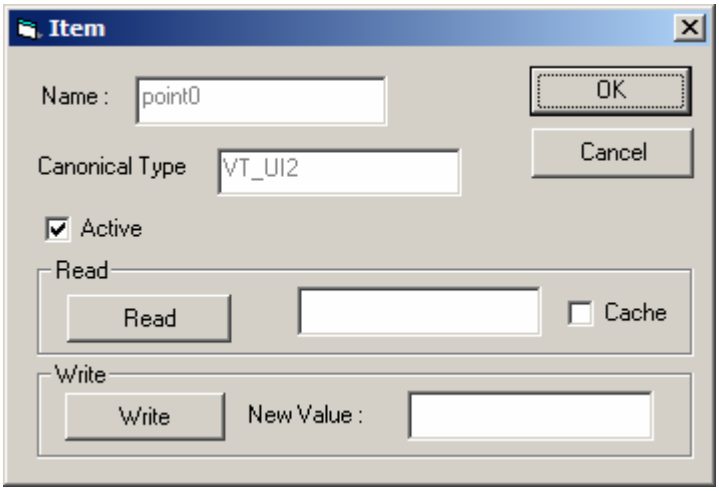

Figure 9 Item Dialog

```
Private Sub Form_Load()
    If Not theItem Is Nothing Then
        txtName = theItem.ItemID
        txtType = GetCanonicalType(theItem.CanonicalDataType)
        chkActive.Value = CInt(theItem.IsActive) * -1
    End If
End Sub
```

Code: ItemDlg (ItemDlg.frm), Form_Load

The **OPCItem** object provides methods to read the current value of the item in the server and write a new value to the item. I have included these facilities into this dialog. The read method provided on an **OPCItem** object performs a synchronous read from the server and can be configured to read either from cache or from the device. To read from cache both the group and item should be active, but synchronous read operations directly from the device do not depend on the active state of either the group or item.

The code below shows the read operation:

```
Private Sub cmdRead_Click()
    Dim source As OPCDataSource
    Dim Value As Variant
    If chkCache.Value = 1 Then
        source = OPCDataSource.OPCCache
    Else
        source = OPCDataSource.OPCDevice
    End If
    theItem.Read source, Value
    txtReadVal.Text = Value
End Sub
```

Code: ItemDlg (ItemDlg.frm), cmdRead_Click

The Write operation on the OPCItem is quite straightforward and is independent of the group or item active state:

```
Private Sub cmdWrite_Click()
    theItem.Write (txtWriteVal)
```

```
    End Sub
```

Code: ItemDlg (ItemDlg.frm), cmdWrite_Click

3.14 Asynchronous Read/Write Operations

Although the *synchronous* read and write methods are provided by the **OPCItem** object the equivalent *asynchronous* methods are accessed from the **OPCGroup** object. This is probably to ensure consistency with the subscription and OnDataChange event.

To demonstrate the use of both these methods add two new menu items to the **Items** menu in the Main form called "**Asynchronous Read"** and "**Asynchronous Write"**. Both methods will automatically generate an "on complete" event, so to show that the method has completed add a text box at the bottom of the form. This will be used to display messages. The menu items also need to be enabled when an item is selected. The code to achieve this needs to be placed in the subroutine "**EnableMenuItems**" in the main form (which was discussed earlier).

3.15 Asynchronous Read

I called the Asynchronous Read menu item "**mnuItemAsyncRead**" and the code shown below can be used to execute the method when the menu item is selected:

```
    Private Sub mnuItemAsyncRead_Click()
        Dim Errors() As Long
        Dim handle(1) As Long
        handle(1) = selectedItem.ServerHandle
        Dim CancelID As Long
        Dim TransID As Long
        TransID = listItems.selectedItem.Index
        selectedGroup.AsyncRead 1, handle, Errors, TransID,
CancelID
        If Errors(1) Then
            txtMsg = "Async Read Error : " & selectedItem.ItemID
        End If
    End Sub
```

Code: MainForm (MainForm.frm), mnuItemSyncRead

The asynchronous read and write commands allow for reading and writing multiple items to and from the device. The Transaction Identifier provides a handle by which a particular asynchronous operation can be identified. In this tutorial application we are only reading one item at a time from a local server, so I have used as the Transaction ID the index in the list view of the item to be read. This provides easy update access to the list view when the **AsyncReadComplete** event is received, as shown by the code below. Notice also the use of the text box **txtMsg** to output any errors encountered:

```
    Private Sub selectedGroup_AsyncReadComplete(ByVal TransactionID
As Long, ByVal NumItems As Long, ClientHandles() As Long,
ItemValues() As Variant, Qualities() As Long, TimeStamps() As Date,
Errors() As Long)
        ' update the item in the list view use the transaction id
        Dim li As ListItem
        Set li = listItems.listItems(TransactionID)
```

```
          With li
              .ListSubItems(1).Text = ItemValues(1)
              .ListSubItems(2).Text = GetQualityText(Qualities(1))
              .ListSubItems(3).Text = TimeStamps(1)
          End With
          ' write a message out
          If Errors(1) = 0 Then
              txtMsg = "AsyncRead Complete " & li.Text & " = " &
CStr(ItemValues(1)) & " Quality = " &
                       GetQualityText(Qualities(1))
          Else
              txtMsg = "AsyncRead Complete Error Reading: " & li.Text
          End If
      End Sub
```

Code: MainForm (MainForm.frm), selectedGroup_AsyncReadComplete

In more complex applications where multiple items are read at once it will be necessary to iterate through the arrays returned by the **AsyncReadComplete** event to check the error array and update accordingly.

3.16 Asynchronous Write

The code to perform an asynchronous write operation on an item is slightly more complex since we require a new dialog to enter a new value. This can be achieved by creating a dialog called “AsyncWriteDlg” with the controls shown below.

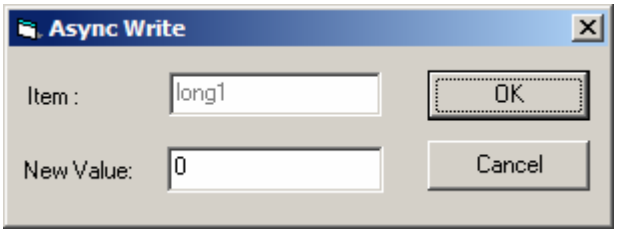

Figure 10 Asynchronous Write Dialog

This dialog is intended purely for obtaining a new value for the item, and is invoked from the mnuItemAsyncWrite menu subroutine, code for which is shown below.

```
      Private Sub mnuItemAsyncWrite_Click()
          Dim Errors() As Long
          Dim handle(1) As Long
          Dim Value(1) As Variant
          Dim TransID As Long
          Dim CancelID As Long
          handle(1) = selectedItem.ServerHandle
          TransID = listItems.selectedItem.Index
          With AsyncWriteDlg
              .txtName = selectedItem.ItemID
              .txtValue = selectedItem.Value
              .Show vbModal
              If .bOK Then
                  Value(1) = .txtValue
                  selectedGroup.AsyncWrite 1, handle, Value, Errors,
TransID, CancelID
                  If Errors(1) Then
                      txtMsg = "Async Write Error : " &
selectedItem.ItemID
```

```
                End If
            End If
        End With
    End Sub
```

Code: MainForm (MainForm.frm), mnuItemAsyncWrite_Click

As you can see most of the code is concerned with setting up the parameters required by the call. The **AsyncWriteComplete** event is handled in the subroutine shown below and simply tests whether the event has any errors. If the item is subscribed to then its properties will be updated in the list view at the requested time interval.

```
    Private Sub selectedGroup_AsyncWriteComplete(ByVal
TransactionID As Long, ByVal NumItems As Long, ClientHandles() As
Long, Errors() As Long)
        Dim li As ListItem
        Set li = listItems.listItems(TransactionID)
        If Errors(1) = 0 Then
            txtMsg = "AsyncWrite Complete For " & li.Text
        Else
            txtMsg = "Async Write Error : " & li.Text
        End If
    End Sub
```

Code: MainForm (MainForm.frm), selectedGroup_AsyncWriteComlete

3.17 Server Shutdown

The Server can request that the clients attached to it disconnect from the server allowing it to shut down in an orderly fashion. The **OCPServer** object generates an event to which the application should respond appropriately. For this tutorial client we will disconnect from the server and clear all the references to objects.

In order to respond to the shutdown event I have created a reference object to the **OPCServer** object " **theServer**" called "**theServerRef**". This is declared in the Main form using the "**WithEvents**" key word:

```
    Public WithEvents theServerRef As opcServer
```

Code: MainForm (MainForm.frm)

This reference is used only to handle the Shutdown Event, the code for which is shown below.

```
    Private Sub theServerRef_ServerShutDown(ByVal Reason As String)
        txtMsg = "Server Shutdown : " & Reason
        Disconnect
    End Sub
```

Code: MainForm (MainForm.frm), theServerRef_ServerShutDown

```
    Private Sub Disconnect()
        theServer.Disconnect
        Set theServer = Nothing
        Set selectedGroup = Nothing
```

```
        Set selectedItem = Nothing
        Set theGroups = Nothing
        RefreshGroupTree
        RefreshItemList
        EnableMenuItems
    End Sub
```

Code: MainForm (MainForm.frm)

4. Conclusion

The OPC Automation Wrapper provides a sensible solution for many OPC applications, naturally occupying the middle ground between the easy-to-use simplicity of the CX-Server OPC Client Control and the full but complex power of the OPC Custom Interface.

It is ideally suited for use as part of Visual Basic based OPC Client applications.