EViews COM Automation

WHITEPAPER AS OF 12/4/2009

EViews COM Automation allows an external program or script to launch and control EViews programmatically and to transfer data back and forth. This allows you to use much of the functionality of EViews within your own programs.

COM Registration

To use EViews COM, you must first register the EViews Type Library on the machine that will need access to EViews. Typically, this is done for you during the EViews installation process. To perform this step manually, first make sure your EViews serial number has been properly registered. Next, open a command window and run EViews with a "/register" command line parameter as follows:

```
C:\Program Files\EViews7\EViews7.exe /register
```

Conversely, to unregister EViews COM, run the EViews program with a "/unregister" parameter. This will remove all relevant entries from the Windows registry. (Note: This does NOT affect EViews serial number registration.)

Adding a Reference

In order to use our EViews COM objects in a VBA or.NET project, you must add a reference to our Type Library.

In Visual Studio, right-click your project file and select "Add Reference…" and then select the "COM" tab. Scroll down to "EViews 1.0 Type Library" and select it. Click OK to add this reference to your project.

Note: If you cannot find "EViews 1.0 Type Library" in the list, this means EViews COM has not been properly registered with Windows. See "COM Registration" above.

In VBA (such as in Excel), switch to the VBA development environment (ALT-F11 in Excel). Under the "Tools" menu, click on "References…". Scroll down to "EViews 1.0 Type Library" and check the associated checkbox. Click OK to close the popup.

In other development environments, please refer to their instructions in properly using COM objects. Our type library definitions are located in a file named "EViewsMgr.dll" in the EViews subdirectory.

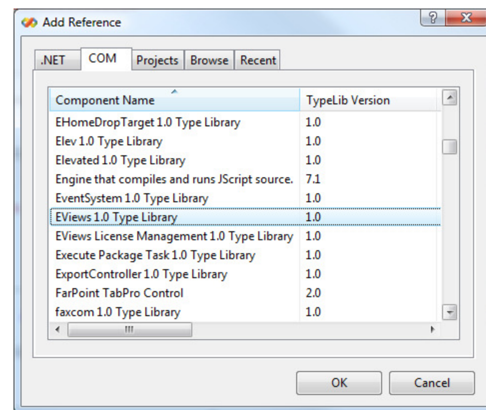

Figure 1 - Visual Studio 'Add Reference' Popup

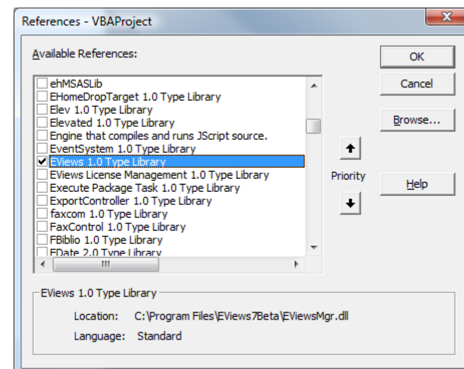

Figure 2 - Excel VBA 'References' Popup

COM Classes

EViews COM is made up of two class objects: Manager and Application.

EViews Manager

The EViews Manager class is used to manage and create instances of the main EViews Application class.

Manager.GetApplication

This method returns an instance of the EViews.Application class. It also allows you to specify whether to create a new instance of EViews or to try and connect to a currently running instance of EViews.

VB.NET Example:

```
Dim mgr As New EViews.Manager
Dim app As EViews.Application
app = mgr.GetApplication(EViews.CreateType.NewInstance)
```

VBA Example:

```
Dim mgr As New EViews.Manager
Dim app As EViews.Application
Set app = mgr.GetApplication(ExistingOnly)
```

Valid parameter values to GetApplication are: `NewInstance` (always create a new instance of EViews), `ExistingOrNew` (look for existing instance of EViews and create another one if not found), and `ExistingOnly` (only look for existing instance, do not create one).

Note: Calling GetApplication is the only way to get a usable Application class object. An Application object that is instanced directly will not be properly initialized and will not be usable.

EViews Application

The EViews Application class provides access to EViews functionality and data. It has the following methods:

```
Show()
Hide()
ShowLog()
HideLog()
Run(commandString)
Lookup(patternString, typeString, returnType)
ListToArray(nameString)
ArrayToList(nameArray)
Get(objectName, naType, naString)
GetSeries(seriesName, sampleString, naType, naString)
GetGroup(seriesNames, sampleString, naType, naString)
GetGroupEx(seriesNames, sampleString, naType, naString, groupOptions)
Put(objectName, objectData, dataType, writeType)
PutSeries(seriesName, seriesData, sampleString, seriesType, writeType)
PutGroup(seriesNames, seriesData, sampleString, seriesType, writeType)
```

Application.Show()

Displays the EViews MDI Frame window. By default, when EViews is started via COM, it is hidden.

Application.Hide()

Hides the EViews MDI Frame window from view.

Application.ShowLog()

Displays the EViews COM Output Log window. Will only be visible if EViews itself is visible.

Application.HideLog()

Hides the EViews COM Output Log window.

Application.Run(commandString)

This method is used to run an EViews command and does not return a value. Some examples of EViews commands include:

VB.NET Example – Opening a workfile:

```
app.Run("wfopen c:\mywork.wf1")
```

Excel VBA Example -- Creating a series:

```
app.Run "series x"
```

Application.Lookup(patternString, typeString, returnType)

Returns a list of object names from the current active workfile that match the specified pattern and/or type. patternString is a required parameter and supports the use of wildcards (e.g. "*" or "g*") and can also specify multiple object patterns that are space delimited (e.g. "g* s*"). typeString is optional and supports all the basic types defined in EViews such as "series", "group", "matrix", etc. You can define multiple types in a space delimited format (e.g. "series group").

The returnType parameter is optional and specifies how to return the list of matching object names:

`LookupReturnString` – returns the names in a single string that is space delimited.
`LookupReturnArray` – returns the names as a 1-dimensional array of strings.
`LookupReturnMatrixAsRows` – returns the names as a 2-dimensional array of strings (1 column, multiple rows). Excel users can apply this return object directly to an Excel range of equal size.
`LookupReturnMatrixAsColumns` – returns the names as a 2-dimensional array of strings (1 row, multiple columns). Excel users can apply this return object directly to an Excel range of equal size.

VB.NET Example:

```
app.Run("wfopen c:\mywork.wf1")
Dim s As String = app.Lookup("s* g*", "series",
       EViews.LookupReturnType.LookupReturnString)
's = "s1 s2 s3 g1 g2 g3"
```

Excel VBA Example:

```
app.Run "wfopen c:\mywork.wf1"
Dim s
s = app.Lookup("s* g*", "series", LookupReturnMatrixAsColumns)
```

```
Dim rows, cols As Integer
rows = UBound(s, 1) - LBound(s, 1) + 1
cols = UBound(s, 2) - LBound(s, 2) + 1
Dim wsht as Worksheet
Set wsht = ActiveSheet
Dim rng
Set rng = wsht.Range(wsht.Cells(1, 1), wsht.Cells(rows, cols))
rng.Value = s 'puts each name into its own cell in the first row
```

Application.ListToArray(nameString)
A utility function to convert a name string (space delimited) into an array of strings (1 column, multiple rows).

Application.ArrayToList(nameArray)
A utility function to convert a name array into a single name string (space delimited).

Application.Get(objectName, naType, naString)
Retrieves data from an object in the current active workfile. The objectName parameter can specify a single object name, or an expression. naType and naString specifies how to return values that are missing in EViews (NA). The allowed values are:

`NATypeAsEmpty` -- returns an empty or blank value
`NATypeAsString` – returns the specified naString value (if specified) in place of the NA
`NATypeAsExcelNA` – returns the Excel NA value (for use in Excel cells)

If not specified, naType defaults to `NATypeAsEmpty`.

For example:

VB.NET Example – Retrieve series "x":
```
Dim o as Object = app.Get("x")
```

Excel VBA Example – Retrieve an equation's covariance matrix:
```
Dim o
O = app.Get("=eq1.@cov")
```

Note: The types of EViews objects that can be returned include series, vectors, matrices, tables, and scalar values.

Application.GetSeries(seriesName, sampleString, naType, naString)
Similar to Get, but restricted to retrieving a series object only. Also supports a named sample or a custom sample string to filter the rows that are returned. naType and naString specifies how to return values that are missing in EViews (NA) (see Application.Get for description of different NATypes).

VB.NET Example – Retrieve series "x" using named sample object "sample1":
```
Dim o as Object = app.GetSeries("x", "sample1")
```

Excel VBA Example – Retrieve series "x" for date range 1980 thru 1990
```
Dim o
```

```
o = app.GetSeries("x", "1980 1990")
```

Application.GetGroup(seriesNames, sampleString, naType, naString)

Similar to GetSeries, but can retrieve multiple series objects as a 2-dimensional array. The seriesNames parameter can be a string with each name delimited by a space, an array of strings, or an Excel Range (either a row or a column of values). This parameter can also make use of wildcards (*) to search for names that fit a pattern. naType and naString specifies how to return values that are missing in EViews (NA) (see Application.Get for description of different NATypes).

VB.NET Example – Retrieve all series objects whose name starts with x or y:

```
Dim o as Object = app.GetGroup("x* y*")
```

Excel VBA Example – Retrieve series "x" and series "y" (along with the date labels) for date range 1980 thru 1990

```
Dim o
o = app.GetGroup("@date x y", "1980 1990", NATypeAsExcelNA)
Dim rows, cols As Integer
rows = UBound(o, 1) - LBound(o, 1) + 1 'number of rows in returned object
cols = UBound(o, 2) - LBound(o, 2) + 1 'number of columns in returned object
Dim wsht as Worksheet
Set wsht = ActiveSheet
Dim rng
Set rng = wsht.Range(wsht.Cells(1, 1), wsht.Cells(rows, cols))
rng.Value = o 'puts data into top left corner of sheet
```

Note: The implicit series "@date" can be included in the list of series names to add a column of date labels to the results.

Application.GetGroupEx(seriesNames, sampleString, naType, naString, groupOptions)

Similar to GetGroup, but allows you to specify a groupOptions string that is a comma-delimited list of options. Valid options include:

`badname` – If a specified object name does not exist in the current workfile, this option will control how EViews will respond. `badname=error` is the default behavior and EViews will return an error on the first object name that doesn't exist. `badname=pad` returns aa empty column padded with `naType` for each object name that is not found.

`transpose` – this option will transpose the 2 dimensional array of data before returning.

VB.NET Example – Retrieve all series object x1, x2, and x3, even if they don't exist in the current workfile, and transpose:

```
Dim o as Object = app.GetGroupEx("x1 x2 x3", , , , "badname=pad,transpose")
```

Application.Put(objectName, objectData, dataType, writeType)

Puts data into an existing or new object in the current workfile. objectData must be in a format that is compatible with the destination object type (if it already exists). For example, if writing to a matrix object, objectData must be a 2 dimensional numeric (an Excel range can also be used as the objectData value). dataType is an optional parameter to manually specify how to read the objectData value (Scalar, Series, Vector, Matrix, etc.). writeType is an optional parameter to specify how to update any pre-existing object with the new data:

`WriteProtect` – If an object already exists with the same name, cancel the Put operation.
`WriteMerge` – Push the source value only if it's not NA. Values outside of source range are left alone.
`WriteMergePreferDestination` – Push the source value only if the destination value is NA. Values outside of source range are left alone.
`WriteUpdate` – (default) Always push the source value (regardless of NA). Values outside of source range are left alone. For series objects, source range is considered the current sample window.
`WriteOverwrite` – Always push the source value (regardless of NA). Values outside of source range are changed to NA.

If not specified, writeType defaults to `WriteUpdate`.

VB.NET Example – Update series "x" with new values:

```
Dim val() as Double = {1.2, 2.3, 3.4, 4.5}
app.Put("x", val, EViews.DataType.DataTypeAuto, EViews.WriteType.WriteMerge)
```

VB.NET Example – Create matrix "m" as a 2x2 matrix:

```
Dim mat(,) as Double = {{1,0}, {0,1}}
app.Put("m", mat, EViews.DataType.DataTypeMatrix,
      EViews.WriteType.WriteOverwrite)
```

Excel VBA Example – Create an alpha series "s" based on the cell values located in specified Excel Range:

```
Dim rng As Range
Set rng = Worksheets(1).Range("D1:D10")
app.Put "s", rng, DataTypeSeriesAlpha
```

Note: DataTypeAuto inspects objectData and defaults to a series object if the dataObject is a 1-dimensional array. Otherwise, it defaults to scalar (for non-arrays) or matrix (for 2 dimensional arrays). For new objects, the first 100 values of the array are inspected to determine if it is numeric or a string.

Application.PutSeries(seriesName, seriesData, sampleString, seriesType, writeType)

Puts data into an existing or new series object in the current workfile. seriesData must be a 1-dimensional array of values. sampleString can be a named sample or a custom sample string which will be used to filter the updates to only those rows that fall in the specified sample. seriesType is an optional parameter to manually specify how to read the seriesData values (Series or Alpha). writeType is an optional parameter to specify how to update any pre-existing object with the new data (see Application.Put for description of different WriteTypes).

VB.NET Example – Update series "x" whose rows fall in the named sample "Sample1" with new values:

```
Dim val() as Double = {1.2, 2.3}
app.PutSeries("x", val, "sample1", EViews.SeriesType.SeriesTypeAuto,
      EViews.WriteType.WriteMerge)
```

Excel VBA Example – Create an alpha series "s" based on the cell values located in specified Excel Range:

```
Dim rng As Range
Set rng = Worksheets(1).Range("D1:D10")
```

```
app.PutSeries "s", rng, "", SeriesTypeAlpha
```

Note: For new objects, SeriesTypeAuto inspects the first 100 elements in the seriesData object to determine if the series is numeric or alpha.

Application.PutGroup(seriesNames, seriesData, sampleString, seriesType, writeType)

Similar to PutSeries, but allows writing to multiple series objects at once. The seriesNames parameter can be a string with each name delimited by a space, an array of strings, or an Excel Range (either a row or a column of values). seriesData must be a 2-dimensional array whose number of columns matches the number of names specified in the first parameter. sampleString can be a named sample or a custom sample string which will be used to filter the updates to only those rows that fall in the specified sample. seriesType is an optional parameter to manually specify how to read the seriesData values (Series or Alpha). writeType is an optional parameter to specify how to update any pre-existing object with the new data (see Application.Put for description of different WriteTypes).

VB.NET Example – Update series "x" and "y" whose rows fall in the named sample "sample1" with new values:

```
Dim val() as Double = {{1.2, 2.3}, {4.5, 5.6}}
app.PutGroup("x y", val, "sample1")
```

Excel VBA Example – Create 2 series objects whose names are located in cells a1 and b1, and whose data is in a2:b11:

```
Dim rngHeaders as Range
Set rngHeaders = ActiveSheet.Range("a1", "b1")
Dim rngData As Range
Set rng = ActiveSheet.Range("a2:b11")
app.PutGroup rngHeaders, rngData
```

Licensing Restrictions

EViews COM is available for both Standard and Enterprise Editions.

Web server access to EViews via COM is not allowed. When being run by other windows services or being run remotely via Distributed COM, EViews will limit COM access to a single instance. Please contact QMS to obtain authorization.

Excel Example (Read with Error Handling):

The following sample code defines an Excel VBA macro that can be used to load all series objects found in a specific worksheet. This also shows an example of basic error handling to display any errors that are reported by EViews. To use this macro, create a blank worksheet in Excel, add a reference to "EViews 1.0 Type Library", create the following macro in a Module file, and then run it to see it in action. Be sure to change the hard-coded values before running the macro to your specific workfile:

```
Public Sub GetWorkfile()
    On Error GoTo ErrorHandler

    'hard coded values
    Dim lsPath As String
    lsPath = "c:\mywork.wf1" 'hard coded to read from mywork.wf1
    Dim liStartColumn As Integer
```

```
    liStartColumn = 1 'put the first object in column 1
    Dim liHeaderRow As Integer
    liHeaderRow = 1 'put the column header in row 1, data after that
    Dim wsht As Worksheet
    Set wsht = ActiveSheet 'output to the current activesheet

    'open connection to EViews
    Dim mgr As New EViews.Manager
    Dim app As EViews.Application
    Set app = mgr.GetApplication(ExistingOrNew)

    'open the workfile
    app.Run "wfopen " & lsPath

    'get the column headers
    Dim columnHeaders
    columnHeaders = app.Lookup("*", "series", LookupReturnMatrixAsColumns)

    'display the column headers
    Dim colcnt As Integer
    'columns are in 2nd dimension
    colcnt = UBound(columnHeaders, 2) - LBound(columnHeaders, 2) + 1
    Dim rng As Range
    Set rng = wsht.Range(wsht.Cells(liHeaderRow, liStartColumn),
       wsht.Cells(liHeaderRow, liStartColumn + colcnt - 1))
    rng.Value = columnHeaders

    'now get the data...
    Dim seriesData
    seriesData = app.GetGroup(columnHeaders, "@all")

    'display the data
    Dim rowcnt As Integer
    'rows are in 1st dimension
    rowcnt = UBound(seriesData, 1) - LBound(seriesData, 1) + 1
    Set rng = wsht.Range(wsht.Cells(liHeaderRow + 1, liStartColumn),
       wsht.Cells(liHeaderRow + rowcnt, liStartColumn + colcnt - 1))
    rng.Value = seriesData

ExitHandler:
    On Error Resume Next
    Set app = Nothing
    Set mgr = Nothing
    Exit Sub

ErrorHandler:
    Dim ret As VbMsgBoxResult
    ret = MsgBox(Err.Description, vbAbortRetryIgnore Or vbCritical, "EViews
       Error")
    Select Case ret
        Case vbAbort:
            Resume ExitHandler
        Case vbRetry:
            Resume
        Case vbIgnore:
```

```
            Resume Next
        End Select
        Resume ExitHandler
    End Sub
```

Excel Example (Write with Error Handling):

The following sample code defines an Excel VBA macro that does the opposite of the previous Read example. It will push all the data from specific columns from the spreadsheet into a new workfile in EViews and then save it. To use this macro, copy this code into your Excel spreadsheet, and change the hard coded values in the function to point to your specific header and data range:

```
    Public Sub SaveToWorkfile()
        On Error GoTo ErrorHandler

        'hard coded values
        Dim lsPath As String
        lsPath = "c:\mywork2.wf1" 'hard coded to write to mywork2.wf1
        Dim liStartColumn As Integer
        liStartColumn = 1 'column 1 has the first column of data
        Dim liHeaderRow As Integer
        liHeaderRow = 1 'column headers are on row 1, with the data after that
        Dim liColCount As Integer
        liColCount = 15 'number of columns of data to push from excel worksheet
        Dim liRowCount As Integer
        liRowCount = 92 'number of rows to push from excel worksheet
        Dim wsht As Worksheet
        Set wsht = ActiveSheet 'read from the current activesheet

        'open connection to EViews
        Dim mgr As New EViews.Manager
        Dim app As EViews.Application
        Set app = mgr.GetApplication(ExistingOrNew)

        'show the EViews window
        app.Show

        'creates a new undated workfile with the correct number of observations
        app.Run "create u " & CStr(liRowCount)

        'get the column header range
        Dim rngHeaders As Range
        Set rngHeaders = wsht.Range(wsht.Cells(liHeaderRow, liStartColumn),
          wsht.Cells(liHeaderRow, liStartColumn + liColCount - 1))

        'get the data range
        Dim rngData As Range
        Set rngData = wsht.Range(wsht.Cells(liHeaderRow + 1, liStartColumn),
          wsht.Cells(liHeaderRow + liRowCount, liStartColumn + liColCount - 1))

        'now push to EViews as Series objects
        app.PutGroup rngHeaders, rngData

        'now save the new workfile
```

```
    app.Run "wfsave " & lsPath

ExitHandler:
    On Error Resume Next
    Set app = Nothing
    Set mgr = Nothing
    Exit Sub

ErrorHandler:
    Dim ret As VbMsgBoxResult
    ret = MsgBox(Err.Description, vbAbortRetryIgnore Or vbCritical, "EViews
      Error")
    Select Case ret
        Case vbAbort:
            Resume ExitHandler
        Case vbRetry:
            Resume
        Case vbIgnore:
            Resume Next
    End Select
    Resume ExitHandler
End Sub
```