EViews COM Automation

March 10, 2015

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 without having to display the EViews window itself.

Licensing Restrictions

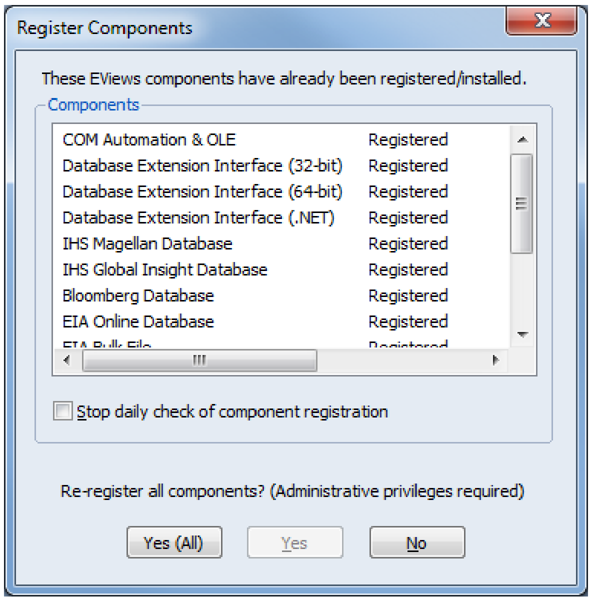
EViews COM Automation is available for both Standard and Enterprise Editions.

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 IHS if you have any questions regarding your license.

Local Registration of EViews COM Automation

In order to control and use EViews via COM, you must first verify that EViews COM Automation is properly registered on your local machine. By default, our EViews installer performs this necessary registration step for you. To verify and perform proper registration, launch EViews and type in the command: REGCOMPONENTS

This will bring up the Register Components dialog. This dialog lists all EViews items that need to be registered on your local machine in order for those features to work properly. "COM Automation & OLE" appears first in the list and also displays whether or not it was been properly registered. If this item does not say "Registered", please click the "Yes (All)" button to perform a full re-registration. You will need to have local admin rights to perform this step.



These EViews components have already been registered/installed.

Components

COM Automation & OLE	Registered
Database Extension Interface (32-bit)	Registered
Database Extension Interface (64-bit)	Registered
Database Extension Interface (.NET)	Registered
IHS Magellan Database	Registered
IHS Global Insight Database	Registered
Bloomberg Database	Registered
EIA Online Database	Registered

☐ Stop daily check of component registration

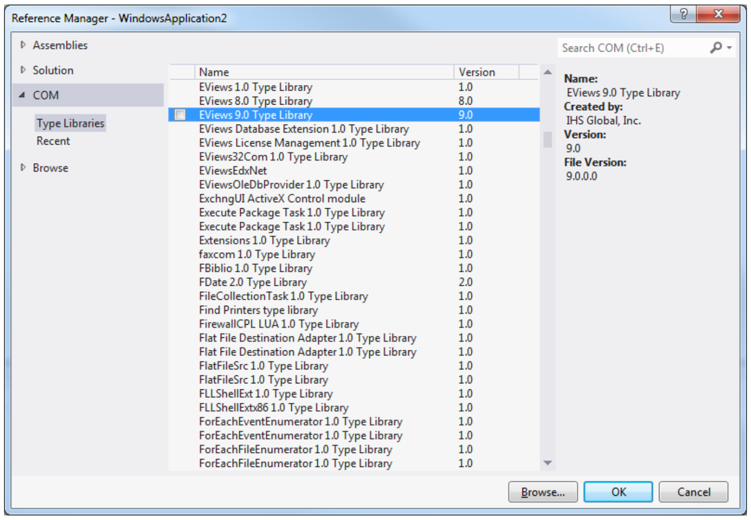
Re-register all components? (Administrative privileges required)

Adding a Reference to the EViews Type Library

The next step is to add a reference to the EViews Type Library to your development project. This allows your development environment to be aware of the various EViews COM objects and their methods.

In this document we'll describe in detail how to use and control EViews from a Visual Studio 2012 VB.NET project and from an Office VBA project (such as Excel 2013) -- but these instructions apply similarly to any other Windows programming environment that allows the use of COM objects (such as C++, VBA Script, MATLAB, etc.).

In Visual Studio, open your project file, then go to the “Project” menu and select “Add Reference…”. On the left, select “Type Libraries” under the "COM" node. You will see a list of available local type libraries appear on the right list. In the list, look for “EViews x.0 Type Library” where x is the version of EViews you are using (1.0 is for EViews 7, 8.0 is for EViews 8, and 9.0 is for EViews 9 – see [Choosing an EViews Version]() for more details). Check the associated checkbox and then click OK to save these changes to your project.



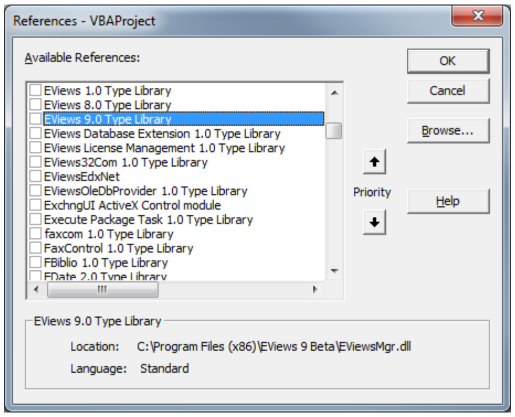
Name	Version
EViews 1.0 Type Library	1.0
EViews 8.0 Type Library	8.0
EViews 9.0 Type Library	9.0
EViews Database Extension 1.0 Type Library	1.0
EViews License Management 1.0 Type Library	1.0
EViews32Com 1.0 Type Library	1.0
EViewsEdxNet	1.0
EViewsOleDbProvider 1.0 Type Library	1.0
ExchngUI ActiveX Control module	1.0
Execute Package Task 1.0 Type Library	1.0
Execute Package Task 1.0 Type Library	1.0
Extensions 1.0 Type Library	1.0
faxcom 1.0 Type Library	1.0
FBiblio 1.0 Type Library	1.0
FDate 2.0 Type Library	2.0
FileCollectionTask 1.0 Type Library	1.0
Find Printers type library	1.0
FirewallCPL LUA 1.0 Type Library	1.0
Flat File Destination Adapter 1.0 Type Library	1.0
Flat File Destination Adapter 1.0 Type Library	1.0
FlatFileSrc 1.0 Type Library	1.0
FlatFileSrc 1.0 Type Library	1.0
FLLShellExt 1.0 Type Library	1.0
FLLShellExtx86 1.0 Type Library	1.0
ForEachEventEnumerator 1.0 Type Library	1.0
ForEachEventEnumerator 1.0 Type Library	1.0
ForEachFileEnumerator 1.0 Type Library	1.0
ForEachFileEnumerator 1.0 Type Library	1.0

Name:
EViews 9.0 Type Library
Created by:
IHS Global, Inc.
Version:
9.0
File Version:
9.0.0.0

Figure 1 - Visual Studio 2012 'Add Reference' Popup

In Excel, open your spreadsheet, then switch to the VBA development environment (press ALT-F11). Under the "Tools" menu, click on "References…". Scroll down to "EViews x.0 Type Library" and check the associated checkbox. Click OK to close the popup.

Note: If you cannot find any version of the EViews Type Library in the list, this means EViews COM Automation has not been properly registered with Windows. Run EViews and type in REGCOMPONENTS, then click the “Yes (All)” button to re-register (see [Local Registration of EViews COM Automation]()).



- [ ] EViews 1.0 Type Library
- [ ] EViews 8.0 Type Library
- [ ] EViews 9.0 Type Library
- [ ] EViews Database Extension 1.0 Type Library
- [ ] EViews License Management 1.0 Type Library
- [ ] EViews32Com 1.0 Type Library
- [ ] EViewsEdxNet
- [ ] EViewsOleDbProvider 1.0 Type Library
- [ ] ExchngUI ActiveX Control module
- [ ] Execute Package Task 1.0 Type Library
- [ ] faxcom 1.0 Type Library
- [ ] FaxControl 1.0 Type Library
- [ ] FBiblio 1.0 Type Library
- [ ] FDate 2.0 Type Library

EViews 9.0 Type Library
Location: C:\Program Files (x86)\EViews 9 Beta\EViewsMgr.dll
Language: Standard

Figure 2 - Excel 2013 VBA 'References' 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" located in the EViews subdirectory.

Creating an Instance of the EViews Application Object

Every time EViews is launched by a user, an associated EViews Application object is created internally to virtually represent that running instance of EViews. All EViews COM functionality is accessed through this Application object and all external COM clients and programs (such as a VB.NET program or Excel VBA) need to use this Application object in order to exchange EViews data, run EViews commands, etc.

But in order to get an instance of the Application object, you must first get an instance of the EViews Manager object.

EViews Manager Object

The EViews Manager class is used to create new instances of EViews, connect to any current instance of EViews already running, or to connect to a specific EViews instance by specifying its Process ID.

Manager.GetApplication

This method returns an instance of the EViews.Application class. It also allows you to specify whether to create a new instance (by launching a new instance of EViews) or to try and connect to any instance of EViews that is already running.

VB.NET Example:

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

VBA Example:

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

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).

By default, a new instance of the EViews application is created without any visible windows. To display the EViews frame window, use the Application object's Show method.

Manager.GetApplicationByProcess

This method returns an instance of the EViews.Application class. If you specify a Process ID (PID), it will attempt to connect to that instance and retrieve that specific Application object. If PID is not specified, this method will look up its current PID and use that to look for a match (finds the instance of EViews that was already created by your program).

VB.NET Example:

```
        Dim mgr As New EViews.Manager
        Dim app As EViews.Application = mgr.GetApplicationByProcess(21)
```

VBA Example:

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

The PID for running applications can be found by running the Windows Task Manager and displaying the PID column in the Processes tab view. Go to the View menu->Select Columns to display this column.

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

EViews Application Object

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)
Get2D(objectName, naType, naString)
GetSeries(seriesName, sampleString, naType, naString)
GetSeries2D(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 main EViews window. By default, when a new instance of EViews is started via the Manager.GetApplication method, it is hidden.

Application.Hide()
Hides the main EViews 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.Get2D(objectName, naType, naString) ***(only exists in EViews 8.0 Type Library and later)***

Similar to Get, but returns all data as a 2-dimensional array, which is especially useful when applying these values to an Excel Range Value property.

Excel VBA Example – Retrieve series "x" and place into an Excel range:
```
Dim o
o = app.Get2D("x")
Dim rng as Range
Dim wsht as Worksheet
Set wsht = ActiveSheet
Set rng = wsht.Range(wsht.Cells(1, 1), wshet.Cells(11, 1))
rng.Value = o
```

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.GetSeries2D(seriesName, sampleString, naType, naString)
(only exists in EViews 8.0 Type Library and later)

Similar to GetSeries, but returns all data as a 2-dimensional array, which is especially useful when applying these values to an Excel Range Value property.

Excel VBA Example – Retrieve series "x" for date range 1980 thru 1990 and place into an Excel range:
```
Dim o
o = app.GetSeries2D("x", "1980 1990")
Dim rng as Range
Dim wsht as Worksheet
Set wsht = ActiveSheet
Set rng = wsht.Range(wsht.Cells(1, 1), wshet.Cells(11, 1))
rng.Value = o
```

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
```

Choosing an EViews Version

On systems that have multiple versions of EViews installed, the choice of which version to instantiate and use is mainly controlled by which version of the EViews Type Library is referenced in your project (assuming you create the Manager object as in our examples above -- see EViews Manager).

Since COM Automation was introduced in EViews 7, only those versions 7 and above are relevant for this discussion. Each subsequent version of EViews was released with a different type library name, Manager Prog ID, and Application Interface:

Version	Type Library Name	Manager Prog ID	Application Interface
EViews 7	EViews 1.0 Type Library	EViews.Manager.1	IApplication
EViews 8	EViews 8.0 Type Library	EViews.Manager.8	IApplication8
EViews 9	EViews 9.0 Type Library	EViews.Manager.9	IApplication9

If your program uses the “EViews 1.0 Type Library”, it will normally only use EViews 7. Likewise, if you reference the 8.0 Type Library, it will only use EViews 8 (again, this is assuming you create the Manager object as in our examples above – see EViews Manager).

Using the Latest Version of EViews

If you want your program to use whichever version of EViews was last installed/registered, you can do this by using the base “EViews 1.0 Type Library”, but changing the way the Manager object is created:

VB.NET Example – Using CreateObject to pick the latest version of EViews that was registered:

```
        Dim mgr As EViews.Manager = CreateObject("EViews.Manager")
        Dim app As EViews.Application = mgr.GetApplication()
        app.Show()
```

Excel VBA Example – Using CreateObject to pick the latest version of EViews that was registered:

```
        Dim mgr As EViews.Manager
        Set mgr = CreateObject("EViews.Manager")
        Dim app As EViews.Application
        Set app = mgr.GetApplication()
        app.Show
```

By using CreateObject and the “EViews.Manager” Prog ID, you are asking Windows to pick which specific version of the Manager object to use. Whenever EViews registers itself, it becomes the primary version, even if newer versions exist on the same system. This means that on a system with EViews 7, 8, and 9 installed together, whichever version was last registered (via REGCOMPONENTS) will be the primary.

Checking for Newer Versions of EViews Dynamically

Using the “EViews 1.0 Type Library” as your base ensures that whichever version of the EViews Manager object is brought back by CreateObject, you will be able to use the subsequent Application object (since all newer versions of the EViews Application object supports the 1.0 type library). However, this will limit you to only using those Application methods that were originally defined for EViews 7. EViews 8 Application methods (such as Get2D and GetSeries2D) will not be accessible. To work around this, you must change the base library to “EViews 8.0 Type Library” (or newer) to get access to the newer IApplication8 interface.

When using the “EViews 8.0 Type Library”, a variable declared as “EViews.Application” is automatically mapped to the newer “EViews.IApplication8” interface (in Visual Studio and VBA). But because CreateObject can return an EViews 7 Manager object, we’ll have to explicitly use the older IApplication interface to receive the Application object. Once received, we can query the object for the newer IApplication8 interface (using TypeOf) to see if it comes from an EViews 8 instance and thus support the newer methods.

VB.NET Example – using “EViews 8.0 Type Library” as the base library:

```
        Dim mgr As EViews.Manager = CreateObject("EViews.Manager")
        Dim app7 As EViews.IApplication = mgr.GetApplication
        app7.Show()
        If TypeOf app7 Is EViews.IApplication8 Then
            Dim app8 As EViews.IApplication8 = app7
            app8.Run("create u 10")
            app8.Run("series x=rnd")
            Dim x = app8.Get2D("x")
            app8 = Nothing
        Else
            MsgBox("Must be EViews 7")
        End If
        app7 = Nothing
        mgr = Nothing
```

Excel VBA Example – using "EViews 8.0 Type Library" as the base library:

```
        Dim mgr As EViews.Manager
        Set mgr = CreateObject("EViews.Manager")
        Dim app7 As EViews.IApplication
        Set app7 = mgr.GetApplication()
        app7.Show
        If TypeOf app7 Is EViews.IApplication8 Then
            Dim app8 As EViews.IApplication8
            Set app8 = app7
            app8.Run "create u 10"
            app8.Run "series x = rnd"
            Dim x
            x = app8.Get2D("x")
            Set app8 = Nothing
        Else
            MsgBox "Must be EViews 7"
        End If
        Set app7 = Nothing
        Set mgr = Nothing
```

Using a Specific Version of EViews Dynamically

CreateObject can also be used to ask for a specific version of EViews directly. Instead of using the generic "EViews.Manager" Prog ID, you can use the version specific Prog ID (e.g. "EViews.Manager.8"):

VB.NET Example:

```
         Dim mgr8 As EViews.Manager = CreateObject("EViews.Manager.8")
         Dim app As EViews.Application = mgr8.GetApplication()
         app.Show()
```

Excel VBA Example:

```
        Dim mgr8 As EViews.Manager
        Set mgr8 = CreateObject("EViews.Manager.8")
        Dim app As EViews.Application
        Set app = mgr8.GetApplication()
        app.Show
```

Special Licensing Notes on 32-bit to 64-bit

The use of EViews COM Automation across different "bit" lines is supported. This means a 32-bit COM client can use 64-bit EViews COM Automation and vice-versa. However, due to the way Windows implements this, this results in a licensing restriction that can be limiting depending on your EViews license and your scenario.

When a program uses a COM object across "bit" lines, Windows supports this by using Distributed COM technology (DCOM). DCOM is primarily used in scenarios where a COM object installed on a remote server can be used just as if it was installed locally.

We currently restrict DCOM use of EViews Automation to a single instance of EViews. This means if you write a 32-bit program to control 64-bit EViews (or vice-versa), you will only be able to run one instance of your program at a time.

You can avoid this restriction by either installing both 32-bit and 64-bit EViews onto your target system (both can be installed on the same system using the same serial – only available with EViews 8 and above), or compile your program to be the same “bitness” as the installed version of EViews (script users should run their script using the proper 32-bit or 64-bit script engine).

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 x.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
```