October, 2002

Fire VFP code for Office events

Office add-ins let you set up event handlers that fire even when your VFP application isn't running

Products: VFP 7/6, Office XP/2000

By Tamar E. Granor

I recently worked, as part of a team, on an application that needed to closely control users' access to Office documents. All documents in the application (whether Word, Excel, or PowerPoint) are created or opened through the application, then presented to the user for editing. The application logs any changes to them when the changes are saved.

Creating and opening Office documents through a VFP application was a simple task. We just needed to instantiate the right server, if it didn't already exist, and call the appropriate method.

However, logging changes when the document was saved wasn't so straightforward. Our first thought was to use VFP 7's ability to implement interfaces with the EventHandler() function to bind our code to the server's events. (For more on this approach, see "Outlook Update" in the June, 2001 issue or the Advisor Answers column in the August, 2001 issue. For a VFP 6 approach to binding, see Advisor Answers in the July, 2001 issue.) But we quickly realized that it was possible for a user to open a document through our application, then close our application, leaving the Office server open. At that point, any event binding performed in our application would no longer be in force.

So, we needed another way to hook into the Office event handlers. Fortunately, the Office servers themselves provide one. Our solution uses add-ins in the Office applications together with a VFP COM object that performs the actual logging.

Hooking Office events

Word and Excel actually provide two ways to attach code to events: templates and add-ins. PowerPoint doesn't offer a way to hook events using templates; in that case, an add-in is required.

So what is an add-in? It's a special kind of document (in the generic sense of "document") that contains code. Add-ins can be loaded interactively in each of the Office applications (look on the Tools menu) or via Automation. There are quite a few commercial add-ins available for the various Office products. (A Web search for "Office" and "add-in" or any of the individual products plus "add-in" turns up thousands of matches. There are add-ins to perform tasks from project management to statistical analysis to putting photos in a PowerPoint presentation.)

An add-in to handle events needs two components: the actual event handler class, and code to connect the event handler to the server instance. Within the event handler class, you put the code to fire when various application or document events occur.

Both the list of events available and the names used for particular events vary with the server. For example, in our application we needed to run some code when the document closed. Word's event is called DocumentBeforeClose, Excel's is the analogous WorkbookBeforeClose, but PowerPoint's is simply PresentationClose.

Creating the event handler add-in

To create an add-in (or write other VBA code), you work in the Visual Basic Editor (VBE) that's available from the Office products. First, create a new, blank, document. Then, open the VBE by choosing Tools-Macro-Visual Basic Editor from the menu in Word, Excel or PowerPoint.

The first step, once you're in the VBE, is to create the event handler class. If the Project Explorer isn't open (normally, it's docked on the left-hand side under the menu), use View-Project Explorer to open it. (Figure 1 shows the Project Explorer in Excel.) Right-click on the project for your new document ("VBAProject (Book 1)" in Figure 1) and choose Insert-Class Module. A new code window opens up. That's where you'll write the event handler code.

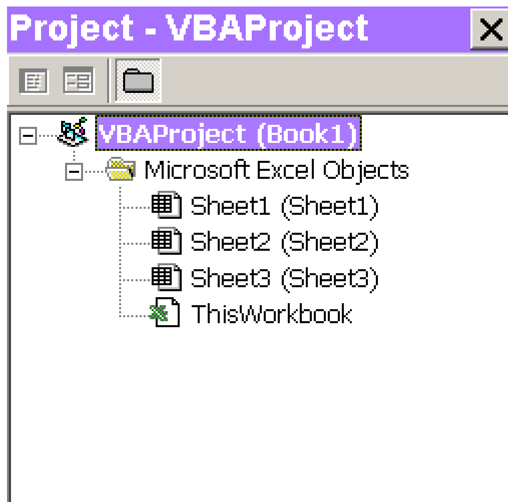

Figure 1. The VBA Project Explorer–This window gives you access to all the projects you're working on in the Visual Basic Editor.

Before starting to write code, it's a good idea to rename the event class. Click on the new class module in the Project Explorer. Then, in the Properties window (normally docked below the Project Explorer), click into the Name property and give your class a meaningful name, like EventHandler

The first step in creating an event handler for one of the Office servers is providing an object reference to the application object. To do so, you need to declare a property to hold it. Switch to the code window, making sure the dropdowns show "(General)" and "(Declarations)," respectively, and type this code:

```
Public WithEvents ExcelApp As Application
```

WithEvents is the magic keyword that lets the object respond to events. Once you've completed this line and pressed Enter, the dropdowns at the top of the code window change. The left-hand dropdown, which shows the available objects, now includes your application property. (See figure 2.) When you choose the property in the left-hand dropdown, the right-hand dropdown is populated with the events to which your code can respond. (See figure 3.)

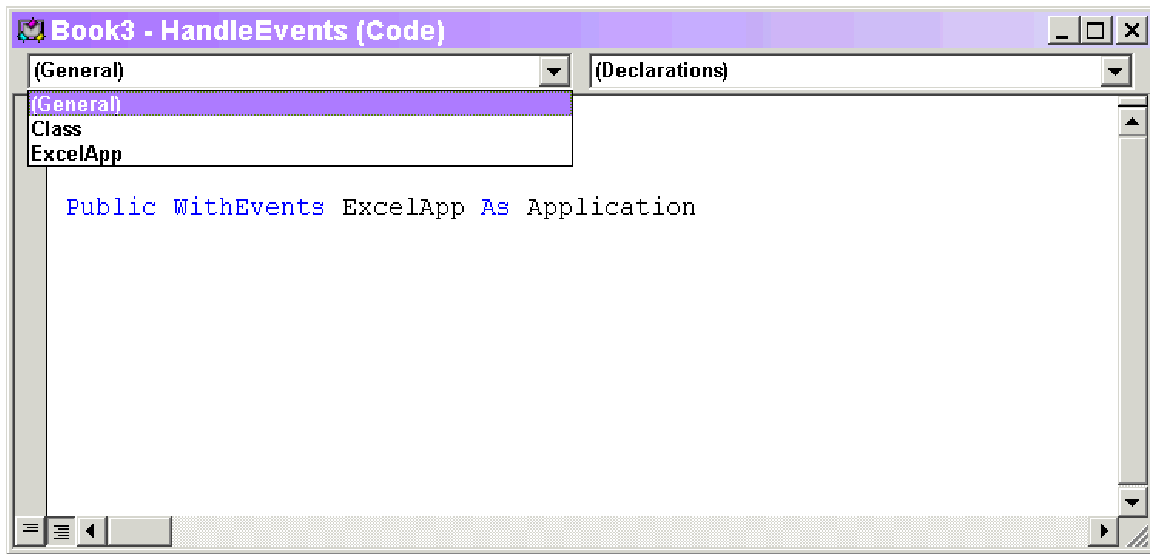

Figure 2. Writing an event handler–Once you declare a variable to hold the application object, that object is accessible in the code window.

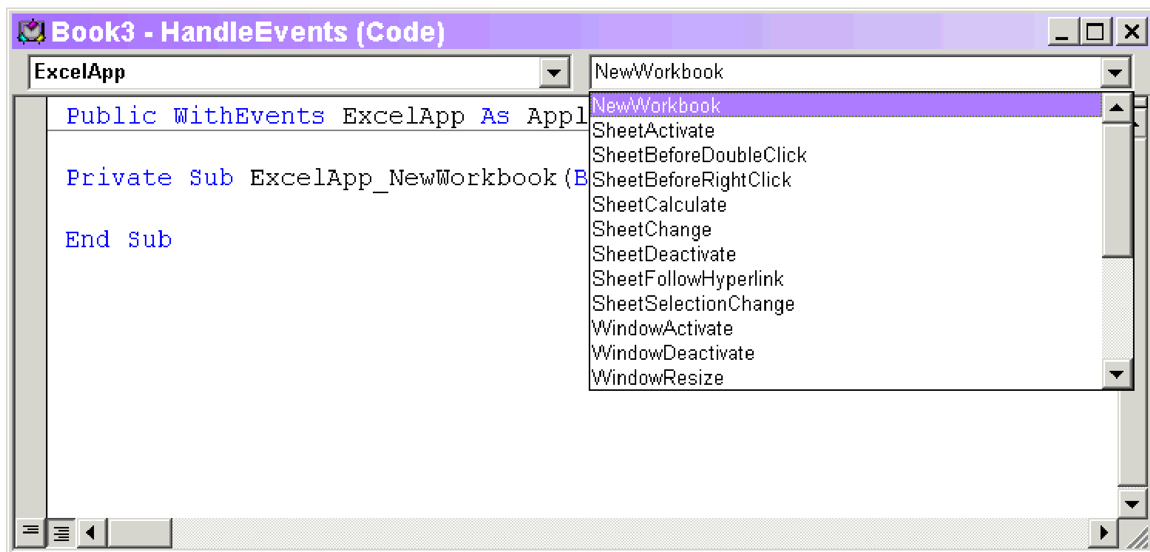

Figure 3. Choosing events–The right-hand dropdown shows the list of available events.

When you choose an event from the right-hand dropdown, stub code for that event is inserted into your code window. (In figure 3, you can see the stub for the NewWorkbook event.)

At this point, you're ready to write the actual VBA code that should run when something happens. Just to demonstrate, let's add code to a few of the events that just displays a messagebox. (As you read the code, keep in mind that the continuation symbol for VBA is "_".)

```
Public WithEvents ExcelApp As Application
```

```
Private Sub ExcelApp_NewWorkbook(ByVal Wb As Workbook)
MsgBox ("New workbook created.")
End Sub

Private Sub ExcelApp_WorkbookBeforeClose( _
   ByVal Wb As Workbook, Cancel As Boolean)
MsgBox ("Closing workbook " & Wb.Name)
End Sub

Private Sub ExcelApp_WorkbookOpen(ByVal Wb As Workbook)
MsgBox ("Opened workbook " & Wb.Name)
End Sub
```

The next step is to connect the actual application object to the defined object reference property (ExcelApp, in the example). To do this, you need to add a module (not a class module) to the project. Again, right-click on the project in the Project Explorer. This time, choose Insert-Module. As before, rename the new module by choosing it in the Project Explorer and changing the Name property in the Properties window. (I called mine "modCatchXL".)

In this new module, declare a variable as a new instance of the object class you just created:

```
Dim cXlHandler As New EventHandler
```

Now create a method whose sole purpose is to connect the application property of the event handler to the application object. I call it HandleEvents. Here's the Excel version of the code:

```
Sub HandleEvents()
Set cXlHandler.ExcelApp = Application
End Sub
```

We need a way to make this code run every time the add-in is loaded. While each of the applications has an event that fires automatically when a document or the application is opened, those events behave differently in different circumstances. (For example, Word doesn't run its AutoExec method when Word was started by automation.) Therefore, I prefer to use code to explicitly run HandleEvents after loading the add-in. That code is shown in "Loading the Add-in" below.

The document now has everything it needs to handle events. The next step is to save it as an add-in. Again, the technique varies with the application. In all cases, though, switch back to the main application (rather than the VBE) before choosing Save As.

Word doesn't have a special add-in file type. Just save the document as a template. If you put it in Word's startup folder (as specified on the File Locations page of Word's Options dialog), the add-in is loaded automatically each time you start Word. Alternatively, you can store it elsewhere and load it manually or via Automation each time you want it.

With Excel and PowerPoint, you need to save the document twice because you can't open an add-in for editing. First, save it as the native format for the application (.XLS for Excel, .PPT for PowerPoint), then use Save As to save it again as an add-in. If you need to modify it later, open the native format version, make the changes, then save it both in the native format and as an add-in.

To save as an add-in in Excel, choose "Microsoft Excel Add-In (*.xla)" from the "Save as type:" dropdown. Once you make this choice, the dialog switches to the AddIns directory for the current user. Saving your add-in in that directory puts it on the list of add-ins available from Excel's Add-Ins dialog, but doesn't automatically enable it. You have to enable it in that dialog or via Automation. If you save it elsewhere, you can load it using the Add-Ins dialog or via Automation. (If you save it in the user's XLStart directory, it will be loaded automatically.)

Saving an add-in in PowerPoint is pretty much the same as for Excel. Once you've saved the presentation, use File-Save As and choose "PowerPoint Add-In (*.ppa)." Again, the dialog switches to the AddIns directory. However, saving your add-in there doesn't have any special advantages except that the dialog for loading an add-in defaults to looking in that directory. Regardless of where it's saved, you can load an add-in through the Add-Ins dialog or via Automation.

Loading the add-in

There are several ways to get your add-in running. In general, it's a two-step process. First, add-ins need to be "registered" to make the application aware of them. Once registered, an add-in can be loaded to get it running. In some cases, you can do both steps at once.

As noted above, in Word, if the add-in is in the Startup directory, it's automatically registered and loaded when you start Word. Otherwise, you have to do something to register and load an add-in.

Most likely, you'll want to use Automation code to get your add-ins loaded, so that you can control them. Each of the applications has an

AddIns collection with an Add method. However, the exact behavior of the Add method varies. In fact, it turns out that using AddIns.Add isn't the best choice in Excel.

In Word, the Add method of AddIns both registers and loads the add-in. So, all you need to do is run the HandleEvents method to create the connection to the application object:

```
oWord = CreateObject("Word.Application")
oWord.AddIns.Add("WordEventHandler.DOT") && Add path
oWord.Run("HandleEvents")
```

Loading and registering an add-in is easy in PowerPoint, too. The Add method registers it, and setting Loaded to True loads it.

```
oPPT = CreateObject("PowerPoint.Application")
oAddIn = oPPT.AddIns.Add("PPTEventHandler") && Add path
oAddIn.Loaded = .T.
oPPT.Run("HandleEvents")
```

Excel doesn't let you register and load an add-in with AddIns.Add unless there's an open workbook. In addition, when you leave an add-in loaded when Excel closes, the next time you run Excel, the add-in shows as installed, but doesn't actually get properly loaded. So a better approach to use with Excel is to open the add-in with the Workbooks.Open method. That registers and loads the add-in cleanly.

```
oXL = CreateObject("Excel.Application")
WITH oXL
   * Add path in next line
   oWorkbook = .Workbooks.Open("ExcelEventHandler.XLA")
   .Run("HandleEvents")
ENDWITH
```

Talking to VFP via a COM object

Now that we have an add-in in which to place code, we want to communicate with FoxPro. There are a number of ways to do this. For example, the VBA code could use ADO to update VFP data directly. Alternatively, the VBA code could instantiate a VFP COM object. Due to the design of our application, the second method was chosen.

Creating COM objects in VFP is not difficult. You need a class defined as OLEPUBLIC, and a project containing the class.

What goes in the class? Methods for whatever you need to do when the Office events fire. In our application, the class has a number of methods, each of which performs a single, fairly simple task. For example, one method updates the document log to indicate that the

document was edited, filling in a timestamp field. We're using a semaphore locking scheme for the documents, so that only one user can edit a given document at a time, so another method releases the semaphore lock when the document is closed.

The best base class for a COM object is Session, which means you'll have to write the code in a PRG file, rather than in the Class Designer. It's a best practice to set up an error handler. Since most COM objects can't have a user interface, a good way to handle errors is to simply log all relevant information to a text file (or a table). In the example below, the Error method handles any errors. However, if your COM object calls on other objects or outside code, and you want unified error handling, you're better off using ON ERROR to set up a global error handler. (Of course, an ON ERROR handler won't be called by any object that has its own error handling code.)

Here's a simple class that has one method to update a log table. The Init method ensures that the log table exists.

```
DEFINE CLASS OfficeResponder AS Session OLEPUBLIC
* COM server to be called in response to Office events.

PROTECTED cLogTable, cLogPath, cLogFullPath
cLogTable = "OfficeLog.DBF"

PROTECTED PROCEDURE Init
* Set up

This.cLogPath = SYS(2023)
This.cLogFullPath = FORCEPATH(This.cLogTable, ;
                              This.cLogPath)

* Make sure the log table exists
IF NOT FILE(This.cLogFullPath)
    CREATE TABLE (This.cLogFullPath) ;
        (cDocument C(50), tModified T)
ENDIF

RETURN

ENDPROC

PROCEDURE UpdateLog( cDocument as String) as Boolean
* Update the activity log

IF VARTYPE(cDocument) <> "C" OR EMPTY(cDocument)
    ERROR 11
    RETURN .F.
ENDIF

INSERT INTO (This.cLogFullPath) ;
```

```
   VALUES (m.cDocument, DATETIME())

RETURN .t.

ENDPROC

PROCEDURE Error(nError, cMethod, nLine)

LOCAL lcErrorMsg, lcFileName

lcErrorMsg = "Error " + TRANSFORM(nError) + SPACE(1) + ;
             CHR(13) + CHR(10) + ;
             "Method " + cMethod + SPACE(1) + ;
             CHR(13) + CHR(10) + ;
             "Line " + TRANSFORM(nLine) + SPACE(1) + ;
             CHR(13) + CHR(10)+ ;
             "At " + TRANSFORM(DATETIME())

************************************************
* Dump an error log into the specified directory
************************************************
lcFileName = FORCEPATH("OfficeResponder.ERR", ;
                       This.cLogPath)
STRTOFILE(lcErrorMsg, lcFileName, .T.)
LIST MEMORY TO FILE (lcFileName) ADDITIVE noconsole
LIST STATUS TO FILE (lcFileName)  ADDITIVE noconsole
RETURN
ENDPROC

ENDDEFINE
```

Once you've created the class, create a project (also called OfficeResponder in the example), add the class to it, and build a .DLL COM server. The resulting .DLL file contains the COM object.

To use the COM object from the Office event handler code, declare an appropriate variable, then use CreateObject() to instantiate the server. Call its methods as needed, and when you're done, set the object variable to Nothing (VBA's version of .null.) to release the server.

In this example, Excel's WorkbookBeforeClose method instantiates the OfficeResponder object and calls the UpdateLog method:

```
Private Sub xlapp_WorkbookBeforeClose(_
   ByVal Wb As Workbook, Cancel As Boolean)
Dim oOfficeResponder

Set oOfficeResponder = _
   CreateObject("OfficeResponder.OfficeResponder")
If Err = 0 Then
    oOfficeResponder.UpdateLog (Wb.Name)
    Set oOfficeResponder = Nothing
End If
```

```
End Sub
```

Be aware that this code fires as you close the workbook that contains it, so you'll get a record in the log for the add-in itself.

Putting it together

We have all the pieces we need now. Just make sure that your main VFP application registers and loads the appropriate add-in at the same time that it instantiates each of the Office applications.

Overall, it takes astonishingly little code to have VFP react when something happens in one of the Office applications. Of course, you can be as creative as you'd like in which events you respond to and the methods called by those events.

This month's Professional Resource CD contains the simple add-ins for Word, Excel and PowerPoint that respond to their respective new document, open document, and close document events. The close document event for each application adds a record to a log table. The PRD also includes the OfficeResponder class, and a program that instantiates each of the servers and loads the appropriate add-in for each.