August, 2001

Advisor Answers

Using Custom Forms in Outlook

VFP 7.0/6.0/5.0/3.0

Q: One of my clients has come up with an Outlook Email requirement I can't seem to figure out. Maybe you can point me in the right direction. I need to execute an Outlook form (EuApp.OFT) from within Visual FoxPro so the user can add some free form data to it before emailing. Part of the boilerplate data comes from tables in my application, which is imported into the form with a form macro.

I've tried using the Outlook Action and Execute properties, but just can't get the right syntax. I can execute it by opening Outlook from the application and then choosing Tools – Forms – Choose Forms from the menu, then choosing User Templates in File System and selecting EuApp.oft. This works, but is not desirable.

–Glenn Laymon (via Advisor.COM)

A: As you know, Outlook is extremely flexible about the way you can enter data for a message or other item. While most users work with the built-in forms, you can design your own forms for collecting data for e-mail messages, appointments, tasks, contacts and most of the other kinds of objects that Outlook works with. This can be very useful in a corporate environment when additional data needs to be stored for each message sent (or appointment made or contact entered, and so forth).

You've indicated the steps to use a custom form interactively. It's actually easier to use one through Automation. Step one is to use the CreateItemFromTemplate method rather than the CreateItem method to instantiate the item. For example, suppose your form is stored in the folder C:\VFPApp\OtherData. To start Outlook and instantiate your form, you'd use code like this:

```
oOutlook = CreateObject("Outlook.Application")
oNS = oOutlook.GetNameSpace("MAPI")
oMsg = oOutlook.CreateItemFromTemplate("C:\VFPApp\OtherData\EuApp.OFT")
```

Now you can fill in the data for the message with Automation, just as you would with a message created with CreateItem. When you're ready for the user to add data, you create an inspector for the form and display it, like this:

```
oInspector = oMsg.GetInspector
oInspector.Activate()
```

An Inspector is a window that displays an Outlook item. When you create a new item interactively, its Inspector is automatically displayed. When you create an item entirely through automation, you don't need an Inspector. But, for those situations where you want to create an item partly through Automation and partly with interaction from the user, you can use the item's Inspector.

Accessing the GetInspector property (though it sounds like a method, it really is a property) returns an object reference to the Inspector for the item; if there is no Inspector, it creates one and returns a reference to it. Then, as you'd expect, the Inspector's Activate method displays the Inspector and allows the user to work with it.

The biggest issue you'll run into when allow the user to interact with Automation code is determining when the user is done. VFP 6 and earlier have no built-in way to respond to events raised by other applications. However, you can use VFPCOM.DLL (available by download from http://msdn.microsoft.com/vfoxpro/downloads/vfpcom.exe) to enable your application to respond to Outlook's events. Since Pamela explained VFPCOM at length in last month's "Advisor Answers," I won't go into details here.

In VFP 7, you don't need VFPCOM to do this. The new IMPLEMENTS keyword allows a class to write code that reacts to events from other classes. Used together with the new EventHandler() function, your VFP code can respond to events in other applications.

The Object Browser in VFP 7 makes it easy to create the class that implements another object's events. Open the type library for the object you want to handle and drag the appropriate interface into a MODIFY COMMAND window. Skeleton code for the class is created, including methods for all the events. You can fill in code for those events to which you want to react.

Outlook uses many different interfaces. (An interface is a set of methods to define a particular behavior.) The interface we're interested in here is called ApplicationEvents and contains a variety of application-level events. The one we want is ItemSend, which fires when an item is sent, whether interactively or by using the Send method. Here's the program created for this interface. (The code that's created by dragging also includes a line to instantiate the class, which isn't shown here. In addition, I've made one change to the code, discussed below.)

```
DEFINE CLASS myclass AS session
  IMPLEMENTS ApplicationEvents IN Outlook.Application
```

```
  PROCEDURE ApplicationEvents_ItemSend(Item AS VARIANT,;
    Cancel AS LOGICAL) AS VOID
    * add user code here
  ENDPROC
  PROCEDURE ApplicationEvents_NewMail() AS VOID
    * add user code here
  ENDPROC
  PROCEDURE ApplicationEvents_Reminder(Item AS VARIANT) ;
    AS VOID
    * add user code here
  ENDPROC
  PROCEDURE ApplicationEvents_OptionsPagesAdd( ;
    Pages AS VARIANT) AS VOID
    * add user code here
  ENDPROC
  PROCEDURE ApplicationEvents_Startup() AS VOID
    * add user code here
  ENDPROC
  PROCEDURE ApplicationEvents_Quit() AS VOID
    * add user code here
  ENDPROC
ENDDEFINE
```

The Object Browser creates a line with the IMPLEMENTS clause that references the type library directly without a path. You can edit it to point to the path on *your* machine. However, it's likely that the type library is installed in a different location on other machines, so if you distribute the application, using the exact path may cause problems. Unlike most Visual FoxPro commands, IMPLEMENTS doesn't look along the path. Therefore, neither leaving the pathless specification nor adding the location to SET PATH would execute the generated program without error. The least troublesome solution, as shown in the code above, is to use the ProgId of the application that provides the events. Alternatively, you can use the type library's GUID.

To react when the user is finished, put code in the ItemSend event to do your post-processing, something like this:

```
PROCEDURE ApplicationEvents_ItemSend(Item AS VARIANT, ;
  Cancel AS LOGICAL) AS VOID
  MESSAGEBOX("Done with message")
  RELEASE this
ENDPROC
```

In setting up the link between Outlook and the event-handling code, you need to think about scope issues. My first instinct was to use a separate program to open Outlook, create the message and open the Inspector. However, without a wait state in that code, the program ends, the Outlook object goes out of scope, and the event bindings are lost. An easy way to solve the problem is to

put the binding code in the same class. Here's a custom method to add to the Event Handling class to get things started:

```
PROCEDURE DoJob
  oOutlook = CREATEOBJECT("Outlook.Application")
  oNS = oOutlook.GetNameSpace("MAPI")
  oMsg = oOutlook.CreateItem(0)
  oInsp = oMsg.GetInspector
  EVENTHANDLER(oOutlook,This)

  oInsp.Activate
ENDPROC
```

To use this class, you just need to instantiate it and call the DoJob method, like this:

```
LOCAL oCreateMessage
oCreateMessage = CreateObject("MyClass")
oCreateMessage.DoJob()
```

At first glance, you might wonder why the object is not released since a local variable is used here. However, Outlook still keeps a reference to the object and VFP doesn't release it until either Outlook or your application terminates. The file OutlookControl.PRG on this month's Professional Resource CD includes all the code for the example class, as well as the instantiation code.

The ability to respond to events from other applications is one of the big new features in VFP 7. It will make it much easier to write various kinds of code. My thanks to Kevin McNeish and Erik Moore, who helped me understand interfaces and the EventHandler() function, and to Christof, who helped me improve the example considerably.

–Tamar