Broaden your Outlook

COM and get it, as Tim Anderson explains how to build COM add-ins in Office 2000.

COM add-ins, a standard way to create extensions for all the Office applications, are a significant new feature of Office 2000. They are particularly welcome in Outlook, which couldn't really be extended with Visual Basic in previous versions, despite the inclusion of VB Script.

In Outlook, the only place to store a script is in a form, which means that it cannot run until the form opens, and customising Outlook 97 or 98 with VB Script is not an option. A COM add-in, on the other hand, can be set to load on startup, giving you full programmatic control over Outlook whenever it is used.

What follows is an explanation of how to build a COM add-in for Outlook, but it is equally applicable to Word, Excel, or any Office 2000 application.

Heart of the matter

At heart, a COM add-in is just another ActiveX DLL. These are not traditional Windows DLLs but code libraries accessed through COM automation.

Not every ActiveX DLL can be a COM add-in. If you've explored COM at all, you'll know that it works through interfaces. Every COM object supports one or more interfaces that tell its clients what it can do.

To be a COM add-in, an object needs to support the interface IDTExtensibility2. Hardened VB hacks might recognise this: it's the same interface used to extend the full VB environment. In other words, COM add-ins are new only to Office.

◀Fig 1 The Add-in Designer in Visual Basic for Applications

It helps to know about IDTExtensibility2. One reason is that you might wonder whether you need to have the Office Developer Edition to create COM add-ins, or whether plain ordinary Office will do. If you want an easy life, get the Office Developer Edition. Along with lots of other bits and pieces, it will let you build COM add-ins from within VBA. It also provides designers that simplify building add-ins, both with VBA and with the full Visual Basic.

Nevertheless, you can create a COM add-in with any tool that can build ActiveX DLLs, including, for example, Visual Basic Professional, without needing the Office Developer Edition. All you need to do is to implement IDTExtensibility2, build the add-in, and register it as an add-in for the target application.

One thing you cannot do without is documentation, which means getting an up-to-date issue of the MSDN (Microsoft Developer Network) library CD.

In and Out

Here's how you might create an add-in for Outlook. This example will keep a log of when emails are sent and received. Outlook has its own journal facility, but creating a custom log means you can organise it in the way you want, or take other actions such as generating an automatic reply to emails. What follows assumes that you have the Office Developer Edition installed.

◀Fig 2 The essential reference for developing Outlook add-ins is the object model, documented in vbaoutl9.chm

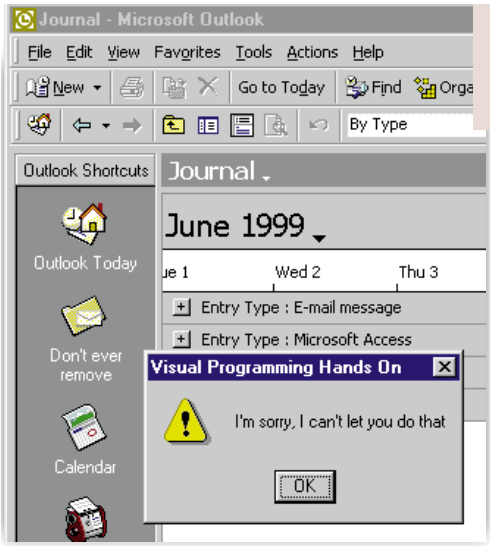
1 In Outlook, display the Visual Basic editor. From the File menu, choose New Project and select Add-in Project from the templates offered. This raises the dialogue in Fig 1. Here you can enter essential details including the name, description, and target application.

One interesting option is the Load Behavior. For something like a logging facility, you would want the add-in always available, so Load On Startup is appropriate. In many cases, though, Load on Demand is the most elegant option. For example, you might install a custom menu that gave access to corporate information, perhaps by querying a database. If Load on Demand is set, then the add-in will not be loaded until the user specifically selects one of its menu options.

Finally, the idea of Load At Next Startup Only is that by loading once, the add-in can customise the target application, adding menus or buttons that call the add-in. Once done, the menus and buttons can be made to persist, but the add-in itself reverts to Load On Demand. There's no need to make any changes on the Advanced tab.

Set a reference to the application you are targeting. This is an option in the Tools — References dialogue. In this case, you will need to find Microsoft Outlook 9.0 Object Library and check it.

2 The next step is to write some code for the add-in. Press F7, or right-click the designer in the Project Explorer, and choose View code. This opens a class module for the add-in. If you drop-down the object box, you will see a built-in



◀FIG 3 THE OFFICIOUS NEW ADD-IN PREVENTS THE REMOVAL OF A SHORTCUT

object called AddinInstance. This is the object that implements IDTExtensibility2 and supports a range of useful events, such as OnConnection and OnBeginShutdown.

Two lines of code will feature in virtually every add-in you create. The idea, remember, is to integrate into the host application, which means you have to both control it programmatically, and respond to its events. To do this, you need to have an object variable that refers to the host application. The OnConnection event is your chance to grab this. The two steps are as follows:

In the General section of the module, declare a global variable for the application object. For example:

```
Private WithEvents ol2k as ↙
Outlook.Application
```

Next, in the OnConnection event handler, write this:

```
Set ol2k = Application
```

where Application is the parameter passed to the OnConnection event handler.

3 Now you can write code that handles Outlook events. For example, here's how you could update a log each time a mail item is sent. In the object box, select **ol2k**, which appeared there when you declared it as a global object. The right-hand event box now has a new range of events, such as NewMail, Quit, and ItemSend.

In this case, ItemSend is the one you need. Select it, and write some code [Example code A]. The snag with this code is that it only tells you when mail was sent. It doesn't specify the recipient, the size of the message, or anything else of interest. To get such information, you need to inspect the Item object. The parameter is of the generic Object type, so to make any sense of it, you need first to set it to a variable of type **mailitem** [Example code B].

There are a couple of points to note here. First, when you need to get a specific interface from a generic object, it's a good idea to use the **If TypeOf...** statement to check that it does in fact support that interface. Second, you'll need to consult the Outlook object model to discover all the properties of an Outlook mailitem. There were 68 of them at the last count, so you have some rich logging options available.

If you consult the Outlook object model, you'll find that it isn't only the top-level Application object that fires events. There's also a **BeforeShortCut Remove** event. The idea is that you can prevent a user from removing a particular ShortCut from the Outlook bar by handling this event. But how do you trap this event, when it isn't one of those fired by the Outlook application object itself?

The answer, once again, is to use an object variable.

The starting point is another global variable:

```
Private WithEvents ↙
olshortcuts as ↙
OutlookBarShortCuts
```

Now you can drop-down the Object box and find this object with its three events — **BeforeShortCutAdd**, **BeforeShortCutRemove**, and **ShortCutAdd**. The **Before ShortCut Remove** event handler has a Cancel parameter, and if you set this to True, then the user will not be able to remove the shortcut in Example code C.

The tricky bit, though, is how you set a reference to the **OutlookBar ShortCuts** object. This requires careful study of the Outlook object model, the object browser, and the skimpy documentation.

Adding this line to the OnConnection event handler works here:

```
Set olShortCuts = ↙
ol2k.ActiveExplorer.Panes(1)↙
.CurrentGroup.Shortcuts
```

Example code A

```
Private Sub ol2k_ItemSend(ByVal Item as Object, ↙
Cancel as Boolean)
Dim FileNum as Integer
FileNum = FreeFile
Open "C:\MAILSENT.LOG" For Append As #FileNum
Print #FileNum, "Sent Mail: " & Str$(Now)
Close #FileNum
End Sub
```

Example code B

```
dim olm as Outlook.MailItem
if not TypeOf Item Is Outlook.MailItem Then
Exit Sub
End If

Set olm = Item
Print #FileNum, "Sent Mail to:" & olm.Recipients(1)...
```

Example code C

```
Private Sub olShortCuts_BeforeShortcutRemove(ByVal shortcut as
OutlookBarShortcut, Cancel as Boolean)
Msgbox "I'm sorry, I can't let you do that"
Cancel = True
End Sub
```

The ↙ symbol in the code segments on these pages denotes that the code continues on the next line.

Here is your host, an IP address in Visual Basic 6

Reader Chris Murray wants to know how to find out the IP address of a given host in Visual Basic 6. He's used Ping.exe, but this requires some detailed parsing of outputs and is prone to problems.

There's an easy way to use the Winsock control, an invisible control that wraps the Windows Sockets API. Set a reference to Microsoft Winsock Control 6.0, and put a Winsock control on a form along with a couple of text boxes called txtHostName and txtIPAddress, and a button.

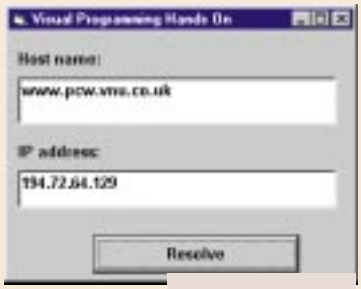
For the button's click handler, add the code in Example D.

Next, in the Winsock control's Connect event, add the code in Example E.

Finally, for the Winsock control's Error event, add the code in Example F.

Now open an internet connection and test the application by entering a host name in the text box and clicking the button. Note that port 80 is for HTTP connections. For FTP. use port 21.

This technique is not quite as good as using ping, because it will fail if the connection on the specified port is refused. It will not work to resolve the IP address of machines on a local network, for example. A better solution would be the gethostbyname API call, although handling the return value is tricky in Visual Basic. This function is part of the full Windows Sockets API.



GETTING AN IP ADDRESS USING THE WINSOCK CONTROL

Example code D

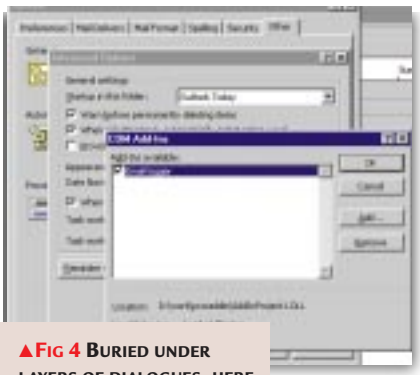
```
If Winsock1.State <> sckClosed then
MsgBox "Connection in use"
Exit Sub
End if
Winsock1.RemoteHost = txtHostName.Text
Winsock1.RemotePort = 80
```

Example code E

```
txtIPAddress.Text= Winsock1.RemoteHostIP
Winsock1.Close
```

Example code F

```
MsgBox str$(Number) + " " + Description
```



▲FIG 4 BURIED UNDER LAYERS OF DIALOGUES, HERE IS THE COM ADD-IN LIST FOR OUTLOOK SHOWING THE NEW ADD-IN

If you get it wrong, you will raise an 'Object variable not set' error. While debugging, it helps to set variables to the intermediate objects. For example:

```
Dim olEx as Outlook.Explorer
Set olEx as ↙
ol2k.ActiveExplorer
```

This way, you can identify which part of a long string of nested objects is causing an error.

4 Once you have made your add-in, the next step is to load it into Outlook. The dialogue for doing so is well hidden. First, display the Tools, Options dialogue, then the Other tab, then the Advanced Options button. Then click the Com Add-Ins button. This displays a list of all the registered add-ins for Outlook. If you use the add-in designer as described above, registration is automatic, so the new add-in will be listed. You can find the list in the registry at HKCU/Software/ Microsoft/ Office/Outlook/Addins, with equivalent locations for the other Office applications.

By default, add-ins can be loaded and unloaded by the user. To prevent an add-in from being unloaded, register it under Hkey_Local_ Machine rather than Hkey_Current_ User. The designer does not offer this as an option, so you would have to register it yourself.

5 In many cases, an add-in will have its own set of menu options or toolbar icons which need to be added to the host application. Without going into this in detail, be warned that it is harder than it looks. The essentials are not difficult: just declare a **CommandBar Button** variable using the WithEvents keyword and add it to the applications CommandBars collection. The place for this code is in the OnConnection event handler.

Then the fun begins. What if the user renames the button using Customize? What if the add-in is set to load on demand, the custom toolbar has persisted from one Outlook session to another, and the user clicks a button before the add-in has loaded?

There are ways to overcome all these and other problems, but dealing with CommandBars is so fiddly that Microsoft has posted a technical guide on the subject on its website.

COM add-ins offer a high degree of integration with the host application. Performance is good, and most things that you can do in a standard Visual Basic application can also be done as a COM add-in. Making sense of the labyrinthine object model can be a strain, but get it right and the results are worth the effort.

PCW CONTACTS

Tim Anderson welcomes your Visual Programming comments and queries. Contact him at visual@pcw.co.uk or via the PCW editorial office.

◆ *Microsoft has some technical papers on programming the Winsock API from Visual Basic. See, for example Q160215, available from support.microsoft.com*