Automation: Interacting with Other Applications

Contents

What is Automation?

In VBA programming the term *Automation* refers to the process of communication between one program and another.

Using Automation you can, for example, take the results of an Access query and write them into a worksheet in an Excel workbook. You can format the cells, rename the worksheet, even get Excel to print out the data, and you will be controlling the process exactly as if you were programming it from within Excel, except you will be working from Access using Automation.

You might want to send some Excel data to someone by email, along with a covering message. With Automation you can program Excel to communicate with Outlook and have it create a new mail message. You can write the data from Excel into the email and add your own message, then send the message. All this can be done from Excel using Automation.

You can use Automation to update slides in a PowerPoint presentation and create new slides as necessary with data from an Excel workbook and an Access Database with a single macro from any one of those programs, then email a copy of the presentation from Outlook along with a customised message.

Automation lets you control one or more programs from another. You have all the programming tools available to you that you would have if you were working within those programs themselves.

The process can take place “invisibly” with the programs opening, working and closing in the background without ever appearing on the screen, or you can open programs so that the user can interact with them and see the result of your commands.

Using Automation exploits the full power of Microsoft Office, enabling all its component applications to work as one.

Client and Server Applications

When two programs communicate with each other using Automation, one is referred to as the *Client* application and the other as the *Server* application.

The *client* application is the one in charge (it is sometimes called the *Controller* application). It is the one that calls up the other application and gets it to do something. It is called the *client* because it is using the services of another application, the *server*. The *server* application is the one that provides its services to the *client*. The *server* responds to instructions and requests from the *client*.

In the examples described above, when Access writes data into an Excel workbook, Access is acting as the *client* application whilst Excel is acting as the *server* application. When Excel sends an email message using Outlook, Excel is the *client* application whilst Outlook is the *server* application.

Early Binding and Late Binding

The terms *Early Binding* and *Late Binding* refer to two different approaches to Automation programming with VBA.

Early Binding

When you use Automation the client application normally creates an *instance* of the server application so that they can talk to each other. With *Early Binding* you set a reference in VBA to the type library of the server application. This means that VBA knows all about the server application, its objects, properties and methods *before* the code is executed. With *Late Binding* no reference is made to the server application's type library, so VBA has no knowledge of the server application until it interacts with it when the code runs.

Early Binding usually creates a new instance of the server application, even if an instance of that application is already running. If the application does not support multiple instances, you should either test for the presence of an existing instance or use *Late Binding*.

For preference you should use *Early Binding*. It is faster than *Late Binding* and, because you will have set a reference to the server application, the Visual Basic Editor will be able to provide the usual help as you type and debug your code.

Late Binding

Late Binding has the advantage that your code does not depend on the existence of the correct object library files being present on the computer on which the code is running. You can develop an application or write a VBA macro for someone else to use on their computer and not need to worry about which version of the server program they are using. (This is a generalisation – it's always better to know these things!).

Late Binding can either make use of an existing instance of the server application or open a new instance. You can use *Late Binding* to open a file in its host application in one step (as opposed to opening the application then using it to open a file, as with *Early Binding*).

How to Tell if Code Uses Early or Late Binding

Most of the examples shown here will use *Early Binding*. VBA procedures using *Early Binding* will contain a code statement near the start of the procedure looking something like this (*Listing 1*):

Listing 1:

```
Dim objXL As New Excel.Application
```

...or like this:

```
Dim objXL As Excel.Application
Set objExcel = New Excel.Application
```

VBA procedures using *Late Binding* will contain code statements near the start of the procedure looking something like this (*Listing 2*):

Listing 2:

```
Dim objXL As Object
Set objXL = CreateObject("Excel.Application")
```

...or like this:

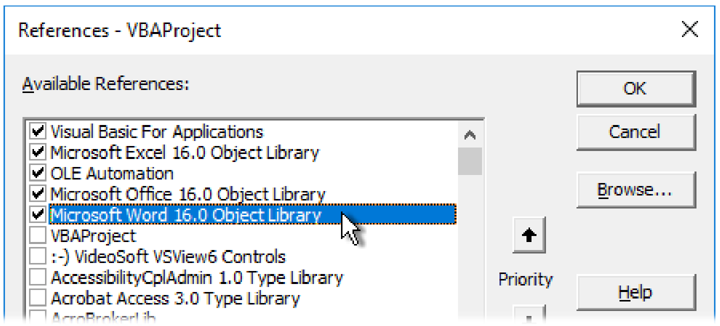
```
Dim objWord As Object
Set objWord = GetObject(, "Word.Application")
```

Setting a Reference to the Server Application

Before writing a procedure using Automation you must make sure that your project contains a reference to the appropriate Object Libraries. Setting a reference to an object library gives your project access to a specific "vocabulary" of code terms so that VBA will "understand" the code you type and be able to provide the necessary programming tools such as Help. If you fail to add the correct references the code compiler will not recognise some of the terms you use and may be unable to run it.

To set a reference to an object library, open the Visual Basic Editor and choose **Tools > References** to open the **References** dialog box. The References dialog box displays a list of all the Object Libraries and Type Libraries available. Libraries to which references are currently set are shown at the top of the list and marked with a tick.

Scroll down the list until you find the reference that you wish to add and place a tick against its name then click the **OK** button. Then next time you open the References dialog you will see that the reference has moved to join the others near the top of the list (*Fig. 1*):



- [✓] Visual Basic For Applications
- [✓] Microsoft Excel 16.0 Object Library
- [✓] OLE Automation
- [✓] Microsoft Office 16.0 Object Library
- [✓] Microsoft Word 16.0 Object Library
- [ ] VBAProject
- [ ] :-) VideoSoft VSView6 Controls
- [ ] AccessibilityCplAdmin 1.0 Type Library
- [ ] Acrobat Access 3.0 Type Library

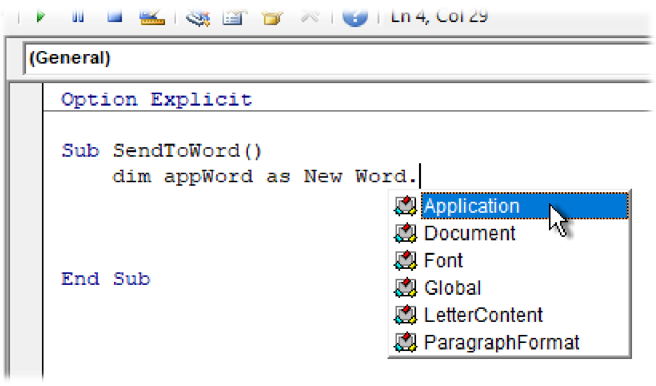
Fig. 1 A reference to the Word 2016 (16.0) Object Library from Excel.

NOTE: When you click on a name in the list it becomes selected, but this does not put a tick in the box! Make sure that you actually click the checkbox when selecting a reference to add.

The Basic Principles of Automation

Create an Instance of the Server Application

Automation begins when you make a link to the server application. Doing this lets the client and server applications start a conversation. As long as you have set a reference to the server application you can use Early Binding and the Visual Basic Editor will be able to help you write your code (*Fig. 2*):



```
Option Explicit

Sub SendToWord()
    dim appWord as New Word.
                            Application
                            Document
                            Font
                            Global
                            LetterContent
                            ParagraphFormat
End Sub
```

Fig. 2 The reference to the Server application lets the Visual Basic Editor provide help.

The resulting statement looks something like this (*Listing 3*) naming the server application and the variable by which it will be referenced in the code:

Listing 3:

```
Sub SendToWord()
    Dim appWord As New Word.Application
```

If you prefer to use Late Binding you can create an instance of the server application using the **CreateObject** method (*Listing 4*):

Listing 4:

```
Sub SendToWord()
    Dim objWord As Object
    Set objWord = CreateObject("Word.Application")
```

To open a file directly into the server application with Late Binding, use the **GetObject** method. When you do this the **GetObject** method asks for two parameters (*[PathName]* and *[Class]*) both of which are optional (but you have to supply at least one). Supply the **PathName** parameter to open an existing file (*Listing 5*):

Listing 5:

```
Sub SendToWord()
    Dim objWord As Object
    Set objWord = GetObject("Z:\Contracts\JobOffer.docx")
```

This method will use the current instance of the server application if there is one open, otherwise it will create one (NOTE: in this case, the object variable refers to the file and not the application.)

To connect to an existing instance of the server application (i.e. one that is already open) without referencing any particular file, use the **GetObject** method and supply the **Class** parameter (*Listing 6*):

Listing 6:

```
Sub SendToWord()
    Dim objWord As Object
    Set objWord = GetObject(, "Word.Application")
```

If there is not an instance of the server application open this method will return an error.

Referring to the Server Application in the Code

When creating an instance of the server application it is assigned a variable name. It is usual to give the variable a name which easily identifies the server application, and to prefix it with something that indicates that it is either an object variable (as in Late Binding) or an application (as in Early Binding). In the examples shown above the instance of Microsoft Word has been named either *objWord* or *appWord*.

Throughout the subsequent macro this variable name must prefix any code statement intended for the server application. This tells VBA that the command is directed at the server application and not the client application. In the example below (*Listing 7*) a program communicates with Excel using the variable name *appXL* to refer to the application:

Listing 7

```
Sub SendToExcel()
    Dim appXL As New Excel.Application
    appXL.Workbooks.Add
    appXL.ActiveSheet.Name = "Contract Data"
    appXL.Range("A1").Select
    'Your code continues here...
End Sub
```

In the next example (*Listing 8*) a program communicates with Word using the variable name *appWord* to refer to the application. Here a *With Statement* has been used to save repeating the variable name on each line. This also improves the efficiency of the code:

Listing 8

```
Sub SendToWord()
    Dim appWord As New Word.Application
    With appWord
        .Documents.Add
        .Selection.Font.Size = 14
        .Selection.TypeText "Contract of Employment"
        .Selection.TypeParagraph
        .Selection.Font.Size = 12
        .Selection.TypeText "Human Resources Department"
        .Selection.TypeParagraph
    End With
    'Your code continues here...
End Sub
```

Doing this also triggers the Visual Basic Editor to display the appropriate help in the form of *Auto List Members* and *Auto Quick Info* relevant to the server application if Early Binding has been employed.

If the prefix is omitted VBA will try to execute the statement on the client application and an error will probably result.

Allowing the User to See the Server Application

Unless a visible instance of the server application already exists and is employed by the Late Binding method, automation works invisibly with the server application opening in the background, doing the tasks assigned to it, and closing without the user being aware of its presence.

On some occasions you may want the server application to appear to the user so that they can see the result of the automation code, and perhaps continue working in it. There are two code statements that are of use here. First, you must make the server application visible to the user by setting its **Visible** property to *True*:

```
appXL.Visible = True
```

Doing this will open the server application's window and place a button on the Windows Taskbar. You will probably also want to bring this window to the front (i.e. activate it) by applying the **AppActivate** method.

```
AppActivate ("Microsoft Excel")
```

The **AppActivate** method can bring any visible application window to the front. You need to supply the application's *Title* as it appears on the *Title Bar* at the top of the application window.

Handing Control of the Server Application to the User

Usually, when a server application has been made visible and activated, and the procedure in the client application has finished, the server application behaves just as if it had been opened by the user. Sometimes this doesn't happen, and the server application fails to respond to the user's commands. To ensure that this isn't the case you can explicitly hand over control of the server application to the user by setting its **UserControl** property to *True*:

```
appXL.UserControl = True
```

This property can also be set to *False* to ensure that, if necessary, the user is prevented from interacting with a visible application.

NOTE: Not all applications allow you to set this property. If you are not sure, debug the procedure (**Debug > Compile** in the Visual Basic Editor) and you will be informed if it is not permitted. If you are not, it is not relevant and won't be a problem.

Closing the Server Application

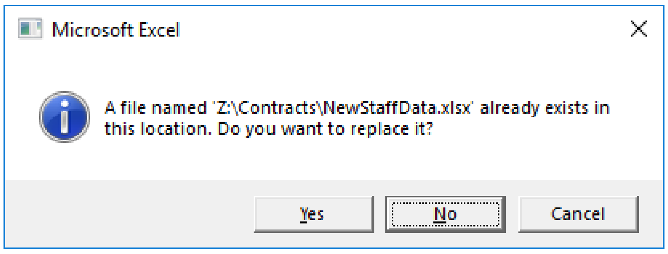
If it isn't necessary to show the server application, remember to include an instruction in the code to close it, saving files where necessary, when you have finished with it (*Listing 9*):

Listing 9:

```
    appXL.ActiveWorkbook.Close _
        SaveChanges:=True, FileName:="C:\Contracts\NewStaffData.xlsx"
    appXL.Quit
End Sub
```

If you fail to quit the application it will remain active in the computer's memory, although not visible to the user, until the computer is shut down. As well as being wasteful of system resources it may interfere with the correct operation of the program when the user tries to open a copy themselves.

When automatically saving files consider whether a file with the same name might already exist. If it does the usual warning message will appear (*Fig. 3*).

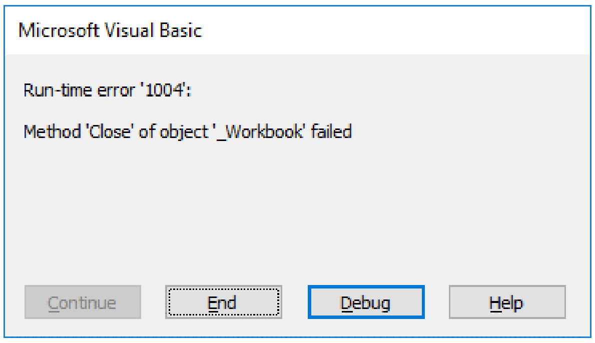


Microsoft Excel

A file named 'Z:\Contracts\NewStaffData.xlsx' already exists in this location. Do you want to replace it?

Fig. 3 The usual warning appears if a duplicate filename exists.

However, if the user decides not to overwrite the existing file and cancels the save operation by choosing **No** or **Cancel** the macro will crash and an error message will be displayed (*Fig. 4*).



Microsoft Visual Basic

Run-time error '1004':

Method 'Close' of object '_Workbook' failed

Fig. 4 Cancelling the close operation caused the macro to crash.

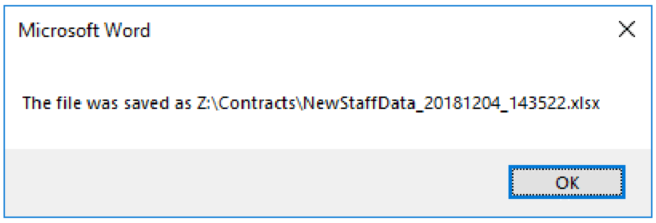
There are different ways to deal with this. You could trap the error with your Error Handler and deal with it appropriately, but this means letting an error happen which is never a good thing and to be avoided if possible. Alternatively, you could prompt the user for a filename.

My preferred method is to generate a unique filename automatically by including a timestamp. This takes the form of a number representing a formatted date detailed to the hour, minute and second, so that even if the user immediately runs the macro a second time a unique filename will be generated (*Listing 10*):

Listing 10:

```
    strFileName = "C:\Contracts\NewStaffData_" & Format(Now, "yyyymmdd_hhnnss") & ".xlsx"
    appXL.ActiveWorkbook.Close SaveChanges:=True, FileName:=strFileName
    appXL.Quit
    MsgBox "The file was saved as " & strFileName
End Sub
```

I then might display a message notifying the user of the full path and filename (*Fig. 5*) so that, if they wish, they can change it manually themselves.



Microsoft Word

The file was saved as Z:\Contracts\NewStaffData_20181204_143522.xlsx

Fig. 5 Notifying the user of the automatically generated filename.

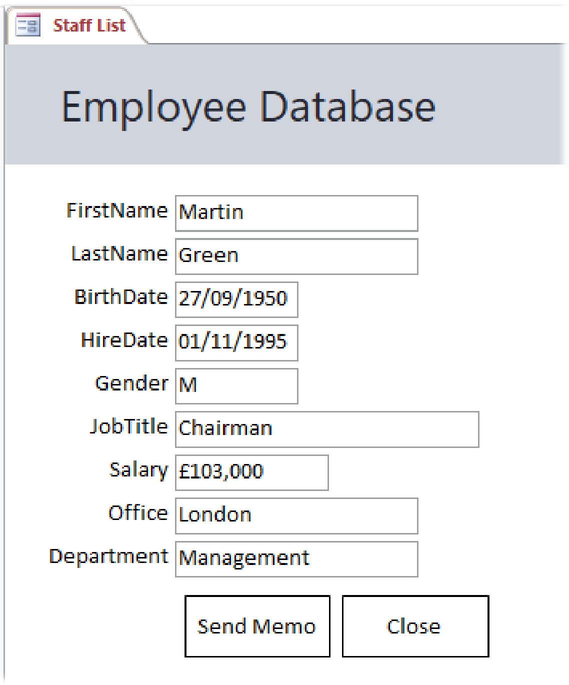
Examples of Automation Code

About the Code Examples

For purposes of clarity these code examples do not all include error handling routines. It is good practice to always include an error handler in your procedures.

Access to Word

In this example, data from the current record on an Access form is used to address a memo in Microsoft Word and runs when the user clicks a command button on the form (*Fig. 6*).

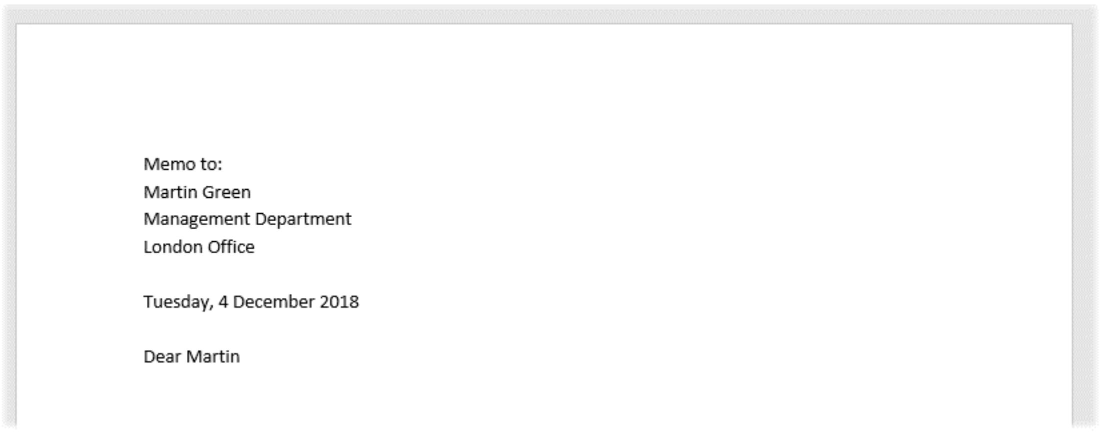


Employee Database

FirstName	Martin
LastName	Green
BirthDate	27/09/1950
HireDate	01/11/1995
Gender	M
JobTitle	Chairman
Salary	£103,000
Office	London
Department	Management

Fig. 6 An Access form with a command button for generating a memo in Word.

Word is then opened showing the document, ready to be edited by the user (*Fig. 7*).



Memo to:
Martin Green
Management Department
London Office

Tuesday, 4 December 2018

Dear Martin

Fig. 7 The memo was generated by Automation code from an Access database.

The code used to create the memo is shown below (*Listing 11*):

Listing 11:

```
Private Sub cmdSendMemo_Click()
    Dim appWord As New Word.Application
    Dim strAddressText As String
    Dim strGreetingText As String
    strAddressText = "Memo to:" & vbCrLf & _
        Me.Firstname.Value & " " & _
        Me.Lastname.Value & vbCrLf & _
        Me.Department.Value & " Department" & vbCrLf & _
        Me.Office.Value & " Office" & vbCrLf & vbCrLf
    strGreetingText = "Dear " & Me.Firstname.Value
    With appWord
        .Documents.Add
        .Selection.TypeText strAddressText
        .Selection.TypeText Format(Date, "dddd, d mmmm yyyy")
        .Selection.TypeParagraph
        .Selection.TypeParagraph
        .Selection.TypeText strGreetingText
        .Visible = True
    End With
End Sub
```

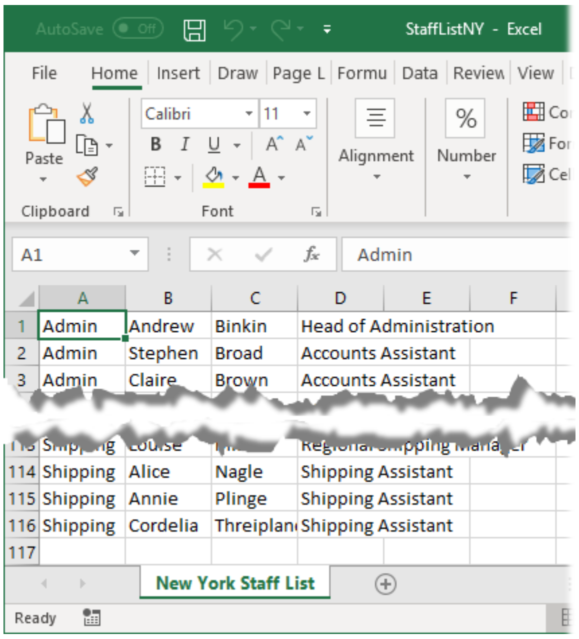
Access to Excel

This example (*Listing 12*) uses automation to write data from an Access table to an Excel worksheet. In fact, Access has much easier ways to do this (e.g. the **TransferSpreadSheet** method) but the example shows how the two programs can interact and offer the programmer complete control over the process.

Listing 12:

```
Sub SendDataToExcel()
    On Error GoTo SendDataToExcel_Err
    Dim cnn As New ADODB.Connection
    Dim rst As New ADODB.Recordset
    Dim appXL As New Excel.Application
    Dim i As Integer
    Set cnn = CurrentProject.Connection
    rst.Open "SELECT * FROM tblStaff " & _
        "WHERE tblStaff.[Office]='New York' " & _
        "ORDER BY tblStaff.[Department], tblStaff.[LastName];" _
        , cnn, adOpenStatic, adLockReadOnly
    appXL.Workbooks.Add
    appXL.ActiveSheet.Name = "New York Staff List"
    rst.MoveFirst
    Do
        With appXL.Range("A1")
            .Offset(i, 0).Value = rst![Department]
            .Offset(i, 1).Value = rst![Firstname]
            .Offset(i, 2).Value = rst![Lastname]
            .Offset(i, 3).Value = rst![JobTitle]
        End With
        i = i + 1
        rst.MoveNext
    Loop Until rst.EOF
    appXL.ActiveWorkbook.Close True, "Z:\Contracts\StaffListNY.xlsx"
SendDataToExcel_Exit:
    On Error Resume Next
    appXL.Quit
    rst.Close
    cnn.Close
    Set rst = Nothing
    Set cnn = Nothing
    Exit Sub
SendDataToExcel_Err:
    MsgBox Err.Description, vbCritical, "Error!"
    Resume SendDataToExcel_Exit
End Sub
```

The code starts by opening a sorted and filtered ADO recordset. It then opens an instance of Excel and a new workbook, then changes the name of the active worksheet. It then loops through the recordset, writing data into the worksheet, moving down a row for each new record. Finally, it saves and closes the workbook and quits Excel. The resulting workbook looks like this (*Fig. 8*):

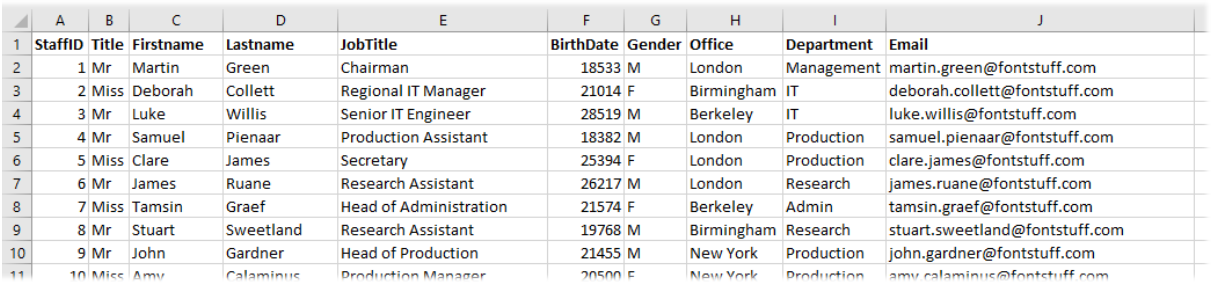


	A	B	C	D
1	Admin	Andrew	Binkin	Head of Administration
2	Admin	Stephen	Broad	Accounts Assistant
3	Admin	Claire	Brown	Accounts Assistant
114	Shipping	Alice	Nagle	Shipping Assistant
115	Shipping	Annie	Plinge	Shipping Assistant
116	Shipping	Cordelia	Threiplan	Shipping Assistant
117				

Fig. 8 An Excel workbook created from Access using Automation.

Excel to Outlook

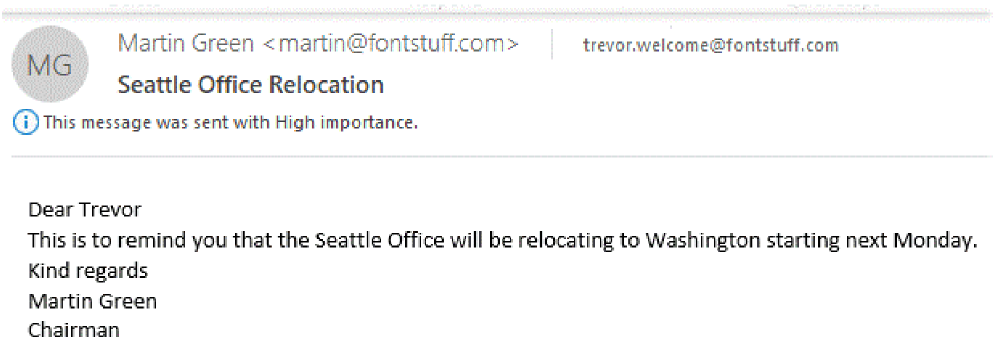
In this example (*Listing 13*) data from an Excel worksheet is used to create email messages in Outlook. A loop is used to move down the rows in a block of data on an Excel worksheet. Each row contains a collection of information about a person, including their name, where they work, and their email address. When a suitable record is found a personalised email is sent to that person by constructing an email message in Outlook. The Excel worksheet containing the source data looks like this (*Fig. 9*):



	StaffID	Title	Firstname	Lastname	JobTitle	BirthDate	Gender	Office	Department	Email
2	1	Mr	Martin	Green	Chairman	18533	M	London	Management	martin.green@fontstuff.com
3	2	Miss	Deborah	Collett	Regional IT Manager	21014	F	Birmingham	IT	deborah.collett@fontstuff.com
4	3	Mr	Luke	Willis	Senior IT Engineer	28519	M	Berkeley	IT	luke.willis@fontstuff.com
5	4	Mr	Samuel	Pienaar	Production Assistant	18382	M	London	Production	samuel.pienaar@fontstuff.com
6	5	Miss	Clare	James	Secretary	25394	F	London	Production	clare.james@fontstuff.com
7	6	Mr	James	Ruane	Research Assistant	26217	M	London	Research	james.ruane@fontstuff.com
8	7	Miss	Tamsin	Graef	Head of Administration	21574	F	Berkeley	Admin	tamsin.graef@fontstuff.com
9	8	Mr	Stuart	Sweetland	Research Assistant	19768	M	Birmingham	Research	stuart.sweetland@fontstuff.com
10	9	Mr	John	Gardner	Head of Production	21455	M	New York	Production	john.gardner@fontstuff.com
11	10	Miss	Amy	Calaminus	Production Manager	20500	F	New York	Production	amy.calaminus@fontstuff.com

Fig. 9 The worksheet containing the source data for the emails.

Each resulting email looks something like this (*Fig. 10*):



Martin Green <martin@fontstuff.com> trevor.welcome@fontstuff.com

Seattle Office Relocation

This message was sent with High importance.

Dear Trevor
This is to remind you that the Seattle Office will be relocating to Washington starting next Monday.
Kind regards
Martin Green
Chairman

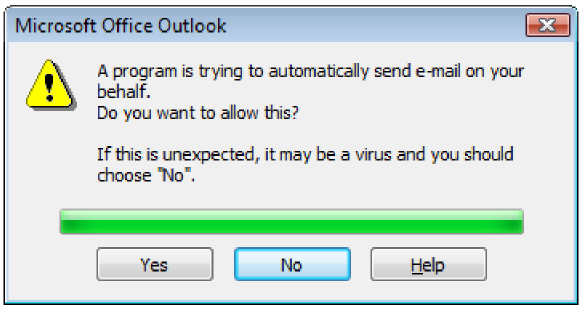
Fig. 10 An email message created in Outlook by Automation from Excel.

Listing 13:

```
Sub SendOutlookEmail()
    On Error GoTo SendOutlookEmail_Err
    Dim appOL As Outlook.Application
    Dim objMailItem As Outlook.MailItem
    Dim i As Integer
    Dim x As Integer
    Dim strMessage As String
    Set appOL = New Outlook.Application
    For i = 1 To Range("A1").CurrentRegion.Rows.Count - 1
        If Range("A1").Offset(i, 7) = "Seattle" Then
            strMessage = "Dear " & Range("A1").Offset(i, 2).Value & vbCrLf & _
                "This is to remind you that the Seattle Office will be " & _
                "relocating to Washington starting next Monday." & vbCrLf & _
                "Kind regards" & vbCrLf & _
                "Martin Green" & vbCrLf & _
                "Chairman"
            Set objMailItem = appOL.CreateItem(olMailItem)
            With objMailItem
                .To = Range("A1").Offset(i, 9).Value
                .Subject = "Seattle Office Relocation"
                .Body = strMessage
                .Importance = olImportanceHigh
                .Send
            End With
            Set objMailItem = Nothing
            x = x + 1
        End If
    Next i
    MsgBox x & " messages were sent.", vbInformation, "Done"
SendOutlookEmail_Exit:
    Set objMailItem = Nothing
    Set appOL = Nothing
    Exit Sub
SendOutlookEmail_Err:
    MsgBox Err.Description, vbCritical, "Error!"
    Resume SendOutlookEmail_Exit
End Sub
```

NOTE: Security measures introduced with Outlook 2002 causes an alert to be displayed each time a VBA macro from any program (including Outlook itself) tries to generate an email message (Fig. 11). This makes sending large numbers of emails this way tedious. Such impediments are put in place in an attempt to prevent the proliferation of "spam" emails.

Remember that if the user cancels the send operation by clicking *No* the macro will crash and an error will result. It is essential therefore to include an error handler in your code as in the example listing.


Microsoft Office Outlook

A program is trying to automatically send e-mail on your behalf.
Do you want to allow this?

If this is unexpected, it may be a virus and you should choose "No".

Fig. 11 A security alert displayed by an older version of Outlook.

Although VBA macros from trusted documents do now not usually prompt such alerts you might wish to find an alternative method for processing emails to large numbers or recipients. One alternative is to send a single email to all the addressees. Do this by building a string containing all the required email addresses, separated by semicolons, and applying it to the *To*, *CC* or *BCC* properties of the message item (remember that a message must have at least one recipient in the *To* field). This would prompt only one warning.

Best practice is to use the *To* field to send the email to, for example, yourself and put the remaining addresses in the *BCC* field to protect the privacy of the recipients. Be aware that there may be a limit to the number of recipients allowed in these fields.

When You Don't Need Automation

Using Built-in Tools

When working with VBA it is easy to forget that many Microsoft Office programs contain built-in tools for collaborating with other Microsoft Office programs for common tasks. These can often be accessed by the user without the need for VBA coding. Most Microsoft Office programs have the facility to send an email message via Outlook. Most also have tools for the import and export of data. These can usually be automated with VBA if required.

Data Exchange with DAO and ADO

Data exchange can often be achieved using DAO (*Data Access Objects* in Access) or ADO (*ActiveX Data Objects* throughout Microsoft Office) providing that you are able to set a reference to one of these object libraries. You would normally expect DAO or ADO to be used when programming *from* Access, but here is an example (*Listing 14*) of sending information in the other direction from Outlook *to* Access. It uses ADO instead of automation to read information from messages in the Outlook Inbox folder and write the data into a table in an Access database.

Listing 14:

```
Sub LogMessages()
    On Error GoTo LogMessages_Err
    Dim ns As NameSpace
    Dim Inbox As MAPIFolder
    Dim Item As MailItem
    Dim cnn As New ADODB.Connection
    Dim rst As New ADODB.Recordset
    Dim i As Integer
    Set ns = GetNamespace("MAPI")
    Set Inbox = ns.GetDefaultFolder(olFolderInbox)
    cnn.Mode = adModeReadWrite
    cnn.Open "Provider= Microsoft.ACE.OLEDB.12.0;" & _
             "Data Source=C:\Contracts\MessageLog.accdb;"
    rst.Open "tblMessageLog", cnn, adOpenKeyset, adLockOptimistic
    For Each Item In Inbox.Items
        rst.AddNew
        rst![From] = Item.SenderName
        rst![To] = Item.To
        rst![Sentdate] = Item.SentOn
        rst![Subject] = Item.Subject
        rst.Update
        i = i + 1
    Next Item
    MsgBox i & " messages were logged.", vbInformation, "Done"
LogMessages_Exit:
    On Error Resume Next
    rst.Close
    cnn.Close
    Set rst = Nothing
    Set cnn = Nothing
    Set Inbox = Nothing
    Set ns = Nothing
    Exit Sub
LogMessages_Err:
    MsgBox Err.Description, vbCritical, "Error!"
    Resume LogMessages_Exit
End Sub
```

NOTE: When ADO connects to a data source such as Access it uses a "connection string", shown here (*Listing 14*) following the statement *cnn.Open…* Connection strings are specific to the type of data source so you must make sure that you use the correct one for your data source. This information is freely available via the Internet.

When Something Goes Wrong

It is possible that when writing and testing your automation code something will go wrong and your code will crash. If you haven't yet included an error handler to tidy things up for you, you are likely to have a hidden instance of the server application still running. Ideally, you should terminate this instance as soon as possible. To do this press **[Control]+[Alt]+[Delete]** on the keyboard to open the Windows **Task Manager** and look through the list of items on the **Processes** tab. Locate the application you wish to terminate, select it and click the **End Process** button to terminate it. Any open (but not hidden) applications should be listed on the *Applications* tab. These can be left running. If you are not sure, **save any open files before you do this!**