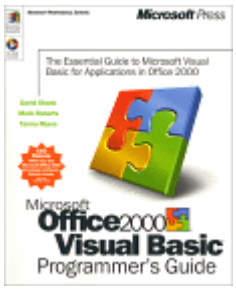
Automation 2000

Your Guide to Microsoft Office 2000 Interoperability

Recommended Reading



Microsoft Office 2000/Visual Basic : Programmer's Guide (Microsoft Professional Editions)
by David Shank, Mark Roberts, Tamra Myers

With detailed technical information delivered straight from the Microsoft Office 2000 documentation team, this practical and precise guide offers hands-on detail for everything from planning and developing Office 2000 solutions, working with data, designing multiuser solutions, and distribution.

Paperback - 800 pages (May 1999)
Microsoft Press; ISBN:1572319526

What is Automation?

Automation (formerly known as OLE Automation) is a feature of the Component Object Model (COM), an industry-standard technology that applications use to expose their objects, methods, and properties to development tools, macro languages, and other applications. For example, a spreadsheet application might expose a worksheet, chart, cell, or range of cells ---each as a different type of object. A word processor might expose objects such as an application, document, paragraph, bookmark, or sentence.

When an application supports Automation, the objects the application exposes can be accessed through Visual Basic. You can use Visual Basic to manipulate the objects by invoking methods or by getting and setting properties of the objects.

In order to understand automation, it is necessary to understand some basic concepts and terminology:

Object	Any item that can be programmed, manipulated or controlled. For example, objects include a textbox, combo box, command button, Word document and more. Microsoft Office 2000 applications include over 500 objects.
Object Property	A property is a characteristic of an object (an adjective). For example, properties of a textbox include: **Name, Visible, ForeColor** and more.
Object Method	An action that you can take on an object (a verb). For example, a method of an Access form is **Close**.
Automation Server	The Automation Server is the application that exposes the automation object(s).
Automation Client	The "Automation Client" is the application that decides which objects to use and when to use them. The "Automation Client" is also referred to as "Automation Container." For example, if a button is clicked on an Access form, and a letter prints in Word: • Automation Server = Word • Automation Client = Access
Binding	Setting the object type to the object variable.
Late Binding	Late Binding occurs when you declare object variables with a specific class and the binding occurs when the code runs (slower).
Early Binding	Early Binding occurs when you declare object variables with a specific class and the binding occurs when you compile the code (faster).

What is Automation?

Automation (formerly known as OLE Automation) is a feature of the Component Object Model (COM), an industry-standard technology that applications use to expose their objects, methods, and properties to development tools, macro languages, and other applications. For example, a spreadsheet application might expose a worksheet, chart, cell, or range of cells ---each as a different type of object. A word processor might expose objects such as an application, document, paragraph, bookmark, or sentence.

When an application supports Automation, the objects the application exposes can be accessed through Visual Basic. You can use Visual Basic to manipulate the objects by invoking methods or by getting and setting properties of the objects.

In order to understand automation, it is necessary to understand some basic concepts and terminology:

Object	Any item that can be programmed, manipulated or controlled. For example, objects include a textbox, combo box, command button, Word document and more. Microsoft Office 2000 applications include over 500 objects.
Object Property	A property is a characteristic of an object (an adjective). For example, properties of a textbox include: **Name, Visible, ForeColor** and more.
Object Method	An action that you can take on an object (a verb). For example, a method of an Access form is **Close**.
Automation Server	The Automation Server is the application that exposes the automation object(s).
Automation Client	The "Automation Client" is the application that decides which objects to use and when to use them. The "Automation Client" is also referred to as "Automation Container." For example, if a button is clicked on an Access form, and a letter prints in Word: • Automation Server = Word • Automation Client = Access
Binding	Setting the object type to the object variable.
Late Binding	Late Binding occurs when you declare object variables with a specific class and the binding occurs when the code runs (slower).
Early Binding	Early Binding occurs when you declare object variables with a specific class and the binding occurs when you compile the code (faster).

An Example of Using an Application Recorder

This demonstrates how to record a new macro in Microsoft Word 2000, and then how to take the generated code and convert it into Automation code for use in Microsoft Access 2000.

1. Start Microsoft Word and select Tools, Macro, Record New Macro.
2. Name the macro "MyMacro" and click OK. You will now see a toolbar appear with a Pause button and a Stop button. We will use this toolbar later when we want to stop recording.
3. From the File menu, click New.
4. Select Blank Document and Click OK. This will open a new blank document.
5. Type in "This is a Macro Recording Test."
6. From the Edit menu, click Select All.
7. Change the font of the selected text to Arial.
8. Click the Stop button on the Macro toolbar.
9. From the Tools menu, click Macro and then click Macros.
10. Click MyMacro and click Edit. This will invoke the Visual Basic Editor and display your recorded macro.

```
Sub MyMacro()

'MyMacro Macro
'Macro recorded on 12/20/98 by John Smith
'
   Documents.Add Template:="", NewTemplate:= False, DocumentType:=0
   Selection.TypeText Text:="This is a Macro Recording Test."
   Selection.WholeStory
   Selection.Font.Name = "Arial"

End Sub
```

11. Copy the code and close the Visual Basic Editor.
12. Close Microsoft Word and do NOT save changes to the document.
13. Start Microsoft Access and create a new module.
14. From the Tools menu, click References and add a reference to the Microsoft Word 9 Object Library.
15. Paste the code into the module. Now all we have to do is change the code so that it will automate Word.
16. We now must add an object variable and use it to create an instance of Microsoft Word. Then, we take the object variable and append to the beginning of each line from the Word macro, as shown below.

```
Sub MyMacro()
'
'MyMacro Macro
'Macro recorded on 12/20/98 by John Smith
'
   'Add the following 3 lines of code to create the instance of
Microsoft
   'Word and make the instance visible
   Dim wordApp As Word.Application

   Set wordApp = CreateObject("Word.Application")
   wordApp.Visible = True

   'Now append the object variable to each line of code which uses a
   'Microsoft Word command.
   With wordApp
      .Documents.Add Template:="", NewTemplate:= False, DocumentType:=0
      .Selection.TypeText Text:="This is a Macro Recording Test."
      .Selection.WholeStory
      .Selection.Font.Name = "Arial"
   End With

End Sub
```

That's it! You're ready to run your Automation procedure from Access.

How to Reference an Automation Object

The following are two functions that can be used to reference an automation object:

- CreateObject — Creates a reference to a new automation object.
- GetObject — Activates an object that has been saved in a file or references a server application that is already running.

The CreateObject Function

The following code uses "CreateObject" to start the Word application and create an instance of the Word application:

```
Dim wordApp as Object
Set wordApp = CreateObject ("Word.Application")
```

The GetObject Function

The GetObject function activates an object that has been saved to disk or references a server application that is already running.

Example to activate a saved file:

```
'Activate a saved file
Dim xlApp as Object
Set xlApp = GetObject("C:\My Documents\MyFile.xls", Excel.Sheet")
```

Example to activate an application that is already running:

```
Activate an application that is already running
Dim xlApp as Object
Set xlApp = GetObject(,"Excel.Application")
```

The New Keyword

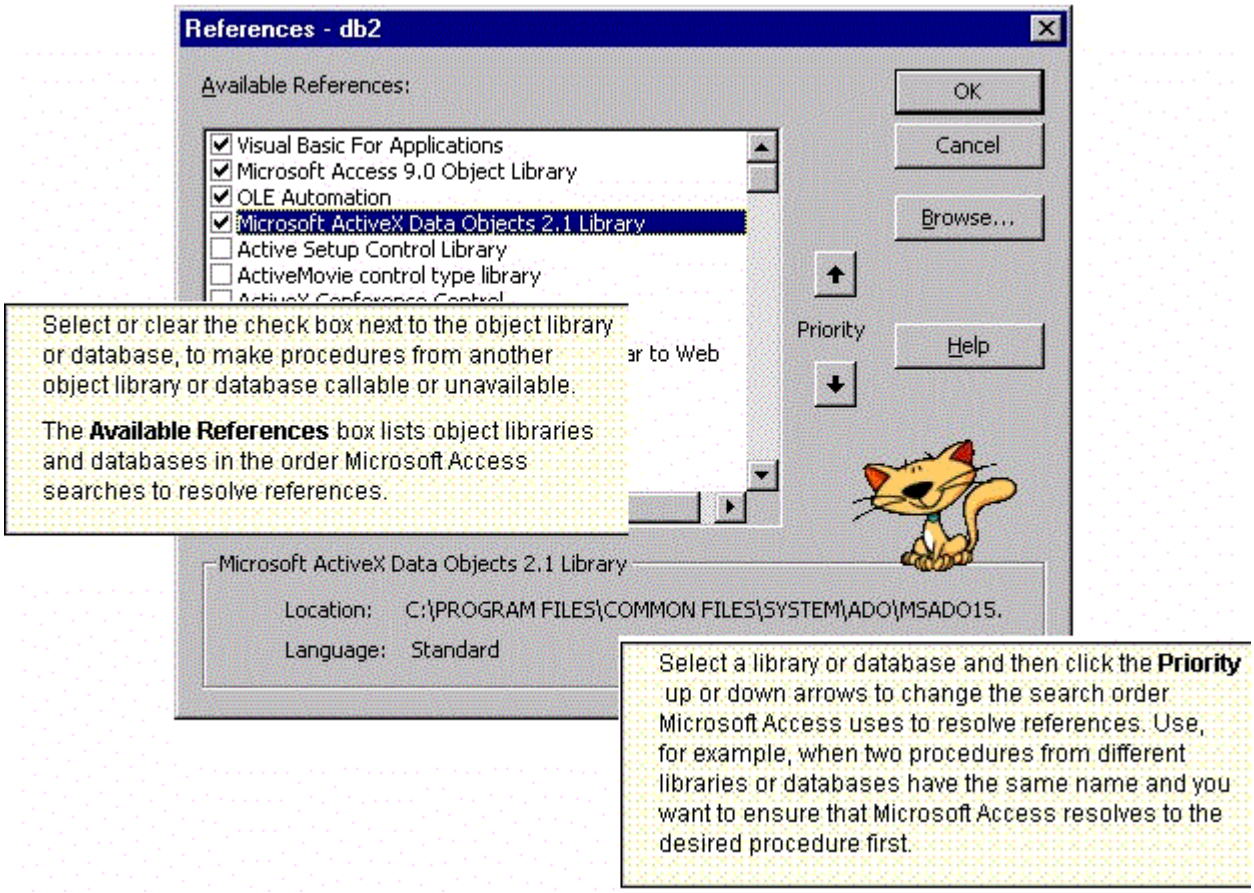
New is a Keyword that enables implicit creation of an object. If you use **New** when declaring the object variable, a new instance of the object is created on first reference to it, so you don't have to use the **Set** statement to assign the object reference.

```
' create a new instance of a Word document object
Dim wordApp as New Word.Document
```

It is not recommended to use the New Keyword when automating Word and Excel. For more information, please download the Knowledgebase article, Q213702 XL2000: Cannot Use New Keyword to Create Workbook Object from http://support.microsoft.com/support/kb/articles/q213/7/02.asp.

Using References

The References dialog box can be used to add or remove libraries from your Visual Basic for Applications project. For instance, if you want to use Automation code to control Microsoft Excel with Early Binding,you would reference the Microsoft Excel 9.0 object library. To view the objects, methods, and properties of this library, you would use the Object Browser.



Early Binding

Early binding declares a variable as a Programmatic Identifier (ProgID) rather than as an Object or a Variant. The variable is initialized by using the CreateObject or GetObject functions; or with the New keyword if both the Automation server and controlling applications support it.

```
Sub ProcessFile()

    Dim xlApp As Excel.Application
    Dim xlBook As Excel.Workbook

    Set xlApp = CreateObject("Excel.Application")
    xlApp.Visible = True
    Set xlBook = xlApp.Workbooks.Open(Filename:="E:\Test\TestFile.CSV")
    xlBook.SaveAs Filename:="E:\Test\TestFile.XLS",
FileFormat:=xlWorkbookNormal
    xlBook.Close
    Set xlSheet = Nothing
    Set xlBook = Nothing
    xlApp.Quit
    Set xlApp = Nothing

End Sub
```

Early binding is the friendly name for what C programmers call Virtual Function Table Binding, or V-Table binding. In order to use Early binding, the controlling application must establish a reference to a type library(.TLB) , object library(.OLB) , or dynamic-link library(.DLL) which defines the objects, methods, and properties of the server application.

Benefits of Early Binding

- Performance: Depending on what your code is doing, early binding may significantly improve the speed of your code.
- Compile-time syntax checking: Syntax errors that you make in Automation code will fail at compile time rather than at run time.
- Code readability: When you declare object variables as specific types, you can simply glance at those declarations to determine what objects a particular procedure uses.
- Viewing objects: When you've set a reference to an application's type library, its objects and their properties and methods show up in the Object Browser. To find out what properties and methods are available for a particular object, just check the Object Browser.
- Getting help: You can get help on another application's object model from the Object Browser, rather than having to launch the application itself.

Late Binding

Late binding declares a variable as an Object or a Variant. The variable is initialized by calling the GetObject or CreateObject functions and specifying the Automation Programmatic Identifier (ProgID). For example:

```
Sub ProcessFile()

    Dim xlApp As Object
    Dim xlBook As Object

    Set xlApp = CreateObject("Excel.Application")
    xlApp.Visible = True
    Set xlBook = xlApp.Workbooks.Open(Filename:="E:\Test\TestFile.CSV")
    xlBook.SaveAs Filename:="E:\Test\TestFile.XLS",
FileFormat:=xlWorkbookNormal
    xlBook.Close
    Set xlSheet = Nothing
    Set xlBook = Nothing
    xlApp.Quit
    Set xlApp = Nothing

End Sub
```

Late binding was the first binding method implemented in Automation controller products. Late binding is the friendly name for what C programmers call IDispatch-based binding. It uses a lot of overhead and is faster than Dynamic Data Exchange (DDE), but slower than Early binding. Late binding is available in all products capable of being an Automation controller.

Using Object Libraries and the Object Browser

When writing automation code, make sure that you add object libraries with the References dialog box. For example, if you want a button in an Access form to create a letter in Word, set a reference to the Microsoft Word 9.0 Object Library. To do this, open a code module in Access. Under the Tools menu, choose References. Click the checkbox for Microsoft Word 9.0 Object Library.

After setting a reference, you can use IntelliSense technology, which provides developers with drop-down menus within the code window for simplified code writing

The Object Browser allows you to view all objects, methods, properties, events, and constants of an Automation server whose type library you have referenced under Tools, References in your Visual Basic for Applications module.

The topmost drop-down list box shows all available libraries you listed in the References dialog box. These libraries allow you to use Early Binding with those Automation servers. The second drop-down list box shows any keywords you have searched on. You can also type a word into this box, and then click the "binoculars" button to search the available libraries for that word. The list box shown immediately under this displays the search results, if any. The list box in the lower left of the dialog box displays all objects in the library. The list box in the lower right of the dialog box displays all methods, properties, events, and constants associated with the selected object in the left list box. The bottom of the dialog displays other information about the currently selected item, such as what kind of object it is, its data type, what arguments it may take and its membership.

OLE Programmatic Identifiers

You can use an OLE Programmatic Identifier (sometimes called a ProgID) to create an Automation object. The following tables list OLE programmatic identifiers for ActiveX controls, Microsoft Office applications, and Microsoft Office Web Components.

ActiveX Controls

To create the ActiveX controls listed in the following table, use the corresponding OLE programmatic identifier.

To create this control	Use this identifier
CheckBox	Forms.CheckBox.1
ComboBox	Forms.ComboBox.1
CommandButton	Forms.CommandButton.1
Frame	Forms.Frame.1
Image	Forms.Image.1
Label	Forms.Label.1
ListBox	Forms.ListBox.1
MultiPage	Forms.MultiPage.1
OptionButton	Forms.OptionButton.1
ScrollBar	Forms.ScrollBar.1
SpinButton	Forms.SpinButton.1
TabStrip	Forms.TabStrip.1
TextBox	Forms.TextBox.1
ToggleButton	Forms.ToggleButton.1

Microsoft Access

To create the Microsoft Access objects listed in the following table, use one of the corresponding OLE programmatic identifiers. If you use an identifier without a version number suffix, you create an object in the most recent version of Access available on the machine where the code is running.

To create this object	Use one of these identifiers
Application	Access.Application, Access.Application.9
CurrentData	Access.CodeData, Access.CurrentData
CurrentProject	Access.CodeProject, Access.CurrentProject
DefaultWebOptions	Access.DefaultWebOptions

Microsoft Excel

To create the Microsoft Excel objects listed in the following table, use one of the corresponding OLE programmatic identifiers. If you use an identifier without a version number suffix, you create an object in the most recent version of Excel available on the machine where the code is running.

To create this object	Use one of these identifiers	Comments
Application	Excel.Application, Excel.Application.9	
Workbook	Excel.AddIn	
Workbook	Excel.Chart, Excel.Chart.8	Returns a workbook containing two worksheets; one for the chart and one for its data. The chart worksheet is the active worksheet.
Workbook	Excel.Sheet, Excel.Sheet.8	Returns a workbook with one worksheet.

Microsoft Graph

To create the Microsoft Graph objects listed in the following table, use one of the corresponding OLE programmatic identifiers. If you use an identifier without a version number suffix, you create an object in the most recent version of Graph available on the machine where the code is running.

To create this object	Use one of these identifiers
Application	MSGraph.Application, MSGraph.Application.8
Chart	MSGraph.Chart, MSGraph.Chart.8

Microsoft Office Web Components

To create the Microsoft Office Web Components objects listed in the following table, use one of the corresponding OLE programmatic identifiers. If you use an identifier without a version number suffix, you create an object in the most recent version of Microsoft Office Web Components available on the machine where the code is running.

To create this object	Use one of these identifiers
ChartSpace	OWC.Chart, OWC.Chart.9
DataSourceControl	OWC.DataSourceControl, OWC.DataSourceControl.9
ExpandControl	OWC.ExpandControl, OWC.ExpandControl.9
PivotTable	OWC.PivotTable, OWC.PivotTable.9
RecordNavigationControl	OWC.RecordNavigationControl, OWC.RecordNavigationControl.9
Spreadsheet	OWC.Spreadsheet, OWC.Spreadsheet.9

Microsoft Outlook

To create the Microsoft Outlook object given in the following table, use one of the corresponding OLE programmatic identifiers. If you use an identifier without a version number suffix, you create an object in the most recent version of Outlook available on the machine where the code is running.

To create this object	Use one of these identifiers
Application	Outlook.Application, Outlook.Application.9

Microsoft PowerPoint

To create the Microsoft PowerPoint object given in the following table, use one of the corresponding OLE programmatic identifiers. If you use an identifier without a version number suffix, you create an object in the most recent version of PowerPoint available on the machine where the code is running.

To create this object	Use one of these identifiers
Application	PowerPoint.Application, PowerPoint.Application.9

Microsoft Word

To create the Microsoft Word objects listed in the following table, use one of the corresponding OLE programmatic identifiers. If you use an identifier without a version number suffix, you create an object in the most recent version of Word available on the machine where the code is running.

To create this object	Use one of these identifiers
Application	Word.Application, Word.Application.9
Document	Word.Document, Word.Document.9, Word.Template.8
Global	Word.Global

Destroying Automation Sessions

An Automation object variable normally is destroyed when it loses scope. For instance, if you declare a local variable to a procedure and set it to an Automation object, that object is destroyed when the procedure ends because the variable's scope was limited only to the procedure. Likewise, an Automation object variable is not destroyed with a Public variable unless that variable is either reinitialized or is explicitly destroyed.

To explicitly destroy an Automation object variable, set it to the keyword 'Nothing' as demonstrated below.

```
Public accessApp As Object

Sub DestroyIt()

   Set accessApp = CreateObject("Access.Application")

   ' do something important...

   ' destroy the object
   Set accessApp = Nothing

End Sub
```

Note that with some Automation Servers, setting an object variable to Nothing doesn't necessarily close the server application. If you're using the CreateObject function to create an instance of an Automation Server, be sure to use a method defined by the Server to quit it if possible and then set the object variable to Nothing. If not, your code may be leaving multiple sessions of the Automation server open. This is very common when automating Microsoft Excel, for instance.

To close the application, use the Quit method prior to setting the object variable out of scope as demonstrated below.

```
Public xlApp As Object

Sub DestroyIt()

   Set xlApp = CreateObject("Excel.Application")

   ' do something important...

   ' quit the application object
   xlApp.Quit

   ' destroy the object
   Set xlApp = Nothing

End Sub
```

Troubleshooting Error 429 When Automating Office Applications

When using the **New** operator or **CreateObject** function in Visual Basic to create an instance of an Office application, you may get the following error:

Run-time error '429': ActiveX component can't create object

This error occurs when the requested Automation object could not be created by COM, and is therefore unavailable to Visual Basic. The error is typically seen on certain computers but not others.

Unlike some errors in Visual Basic, there is no one cause to an error 429. The problem happens because of an error in the application or system configuration, or a missing or damaged component. Finding the exact cause is a matter of eliminating possibilities. If you encounter this error on a client computer, download the Knowledgebase article, Q244264 Troubleshooting Error 429 When Automating Office Applications from http://support.microsoft.com/support/kb/articles/Q244/2/64.ASP.

This article provides some troubleshooting tips to help you diagnose and resolve common problems that are known to cause this error.

Microsoft Access

The Microsoft Office Database Management System

Microsoft Access 2000 makes it easy to get the information you need and provides powerful tools that help you organize and share your database so you and your team make better decisions. Quickly find answers that count, share information over intranets, and build faster and more effective business solutions.

Build Powerful Business Solutions More Easily and Find Answers Faster

Enable Web collaboration and improve productivity with new tools in Access 2000. Make data immediately available to any coworker. Update sales figures from the road or quickly check on customer details. Customize your views and formats to show precisely the information you need. And use built-in Microsoft SQL Server™ integration to create a scalable database that can grow with your business.

Detect if Access is Installed.

As Access is not included in Office Standard or may not be present the user's machine, it may be useful to test for the existence of Access before you launch into your code.

```
Sub existenceCheck()
   Dim objApp As Object
   Dim strNotFound As String
   Const ERR_APP_NOTFOUND As Long = 429

   strNotFound = "Access is not installed on this machine. " _
      & vbCrLf & "Unable to automate Access."

   On Error Resume Next
   ' Attempt to create late-bound instance of Access application.
   Set objApp = CreateObject("Access.Application")

   If Err = ERR_APP_NOTFOUND Then
      MsgBox strNotFound
      Exit Sub
   End If
   With objApp
      ' Code to automate Access here.
      .Quit
   End With
   Set objApp = Nothing
End Sub
```

For applications where you wish to offer the user the functionality of Access Reports, you may wish to utilize the Snapshot viewer if there is the possibility that the user doesn't have Access installed. For more information on the Snapshot viewer see the Knowledgebase article, Q175274 which is located at http://support.microsoft.com/support/kb/articles/Q175/2/74.ASP

Creating an Instance of Microsoft Access

In order to manipulate Microsoft Access objects through Automation, you must create an instance of the Microsoft Access Application object.

You can do this either through Late binding or Early binding.

Late Binding:

```
Dim accessApp As Object
Set accessApp = CreateObject("Access.Application")
```

Early Binding:

```
Dim accessApp As Access.Application
Set accessApp = CreateObject("Access.Application")
```

Early Binding with the 'New' Keyword;

You can also use the 'New' keyword with early binding to create an instance of Microsoft Access. The 'New' keyword allows you to dimension your variable as a new instance of Microsoft Access without having to use either the CreateObject or GetObject functions. As soon as any method or property of the object variable is referred to, the instance is created. For instance,

```
Dim accessApp As New Access.Application
accessApp.Visible = True
```

Referencing a New Instance of Microsoft Access

When using Automation to create a new instance of Microsoft Access, refer to the Application object.

For information on how to reference an existing instance of Microsoft Access, see Referencing an Existing Instance of Microsoft Access.

To open a database in the new instance use the OpenCurrentDatabase method. To create a new database use the NewCurrentDatabase method.

```
Dim accessApp As Access.Application

Sub OpenNorthwind()
   Set accessApp = CreateObject("Access.Application.9")

   With accessApp
      .OpenCurrentDatabase("C:\Program Files\Microsoft
Office\Office\Samples\Northwind.mdb")
   End With

End Sub
```

Referencing an Existing Instance of Microsoft Access

Sometimes it is useful for your code to refer to an already existing instance of Microsoft Access rather than creating a new one. For information on how to reference a new instance of Microsoft Access, see Referencing a New Instance of Microsoft Access. Thus, what is the best way to determine if an instance of Microsoft Access is already running? The best way to determine that is to use the GetObject function with no PathName argument and "Access.Application" as the Class argument.

If an instance of Access exists, then the object variable will refer to the Application object of that instance. If more than one instance of Access exists, one is chosen at random.

If there are no instances of Access, a trappable run-time error will occur. Error handling should be used in your code in case an instance of Access does not exist. Once the reference is created, you can determine which database is open, if any, and use the CloseCurrentDatabase, OpenCurrentDatabase, or NewCurrentDatabase methods as needed.

```
Sub StartAccess()
    On Error Resume Next
    Set objAccess = GetObject(, "Access.Application")

    If Err.Number <> 0 Then
        MsgBox "No instance of Access available.  Starting" _
            & vbCrLf & "a new instance..."
        Err.Clear
        Set objAccess = CreateObject("Access.Application")
    Else
        MsgBox "There is an instance of Access available."
    End If

End Sub
```

Terminating an Instance of Microsoft Access

Normally, the instance of Access is terminated when the object variable that refers to the instance is set to Nothing or loses scope. However, if a table, query, form, or report is still open in normal view or a table is open in design view, the instance will not be terminated when its object variable is set to Nothing or loses scope. In this case, the instance will terminate only after the user closes the opened objects and the instance's UserControl property is false. In any case, if the UserControl property of the instance is true, the instance will not be terminated when its object variable is set to Nothing or loses scope. You must use the Quit method if these conditions exist and you want to terminate the instance. For example:

```
accessApp.Quit
Set accessApp = Nothing
```

However, the Quit method should not be used if the instance is a "special" instance, as described in Calling Microsoft Access Built-in Functions and your application may continue to make direct calls to Access functions. For example, if GetObject retrieves an existing instance of Access that was created automatically by the client because the client application made a direct call to an Access function, that "special" instance of Access should not be terminated by using Quit.

If the instance is terminated by using Quit and the client application makes another direct call to an Access function, the client will generate an "OLE Automation error" if it cannot find the special instance. Use the Quit method only if you're sure the instance is not a special instance or if the application will not continue to make direct calls to Access functions. Note: When the object variable that points to a special instance is set to Nothing or loses scope, it will not terminate the special instance, regardless of its UserControl setting. A "special" instance will be terminated automatically by the client when the client application is closed or reset.

Tip: To prevent an instance of Access from terminating when its object variable loses scope, use a module- or public-level variable to reference that instance of Access rather than a procedure-level variable.

NewCurrentDatabase Method Example

The following example creates a new Microsoft Access database from another application through Automation, and then creates a new table in that database.

You can enter this code in a Visual Basic module in any application that can act as a COM component. For example, you might run the following code from Microsoft Excel, Microsoft Visual Basic, or Microsoft Access.

When the variable pointing to the **Application** object goes out of scope, the instance of Microsoft Access that it represents closes as well. Therefore, you should declare this variable at the module level.

```
' Include following in Declarations section of module.
Dim appAccess As Access.Application

Sub NewAccessDatabase()
        Dim dbs As Object, tdf As Object, fld As Variant
        Dim strDB As String
        Const DB_Text As Long = 10
        Const FldLen As Integer = 40

        ' Initialize string to database path.
        strDB = "C:\My Documents\Newdb.mdb"

        ' Create new instance of Microsoft Access.
        Set appAccess = CreateObject("Access.Application.9")

        ' Open database in Microsoft Access window.
        appAccess.NewCurrentDatabase strDB

        ' Get Database object variable.
        Set dbs = appAccess.CurrentDb

        ' Create new table.
        Set tdf = dbs.CreateTableDef("Contacts")

        ' Create field in new table.
        Set fld = tdf. CreateField("CompanyName", DB_Text, FldLen)

        ' Append Field and TableDef objects.
        tdf.Fields.Append fld
        dbs.TableDefs.Append tdf
        Set appAccess = Nothing
End Sub
```

Output an Access Report

This function outputs a report in the format specified by the optional **lngRptType** argument. If lngRptType is specified, the report is automatically opened in the corresponding application. lngRptType can be any of the following constants defined by Enum opgRptType in the Declarations section of the module:

```
Enum opgRptType
   XLS = 1
   RTF = 2
   SNAPSHOT = 3
   HTML = 4
End Enum
```

Where

- XLS = output to Excel
- RTF = output to Rich Text Format
- SNAPSHOT = output to Access snapshot report format
- HTML = output to HTML

If lngRptType is not specified, the report is opened in Access and displayed in Print Preview.

```
Function GetReport(Optional lngRptType As opgRptType) As Boolean

    Dim acApp As Access.Application
    Dim strReportName As String
    Dim strReportPath As String

   Const SAMPLE_DB_PATH As String = "c:\program files\" _
      & "microsoft office\office\samples\northwind.mdb"

   strReportName = "Alphabetical List of Products"
   strReportPath = Options.DefaultFilePath(wdDocumentsPath) & "\"

   ' Start Access and open Northwind Traders database.
   Set acApp = GetObject(SAMPLE_DB_PATH, "Access.Application")
   With acApp
      ' Output or display in specified format.
      With .DoCmd
         Select Case lngRptType
            Case XLS
               .OutputTo acOutputReport, strReportName, _
                  acFormatXLS, strReportPath & "autoxls.xls", True
            Case RTF
               .OutputTo acOutputReport, strReportName, _
                  acFormatRTF, strReportPath & "autortf.rtf", True
               ' Snapshot Viewer must be installed to view snapshot
               ' output.
            Case SNAPSHOT
               .OutputTo acOutputReport, strReportName, _
                  acFormatSNP, strReportPath & "autosnap.snp", True
            Case HTML
               .OutputTo acOutputReport, strReportName, _
                  acFormatHTML, strReportPath & "autohtml.htm", _
                  True, "NWINDTEM.HTM"
            Case Else
               acApp.Visible = True
               .OpenReport strReportName, acViewPreview
         End Select
      End With
      ' Close Access if this code created current instance.
```

```
      If Not .UserControl Then
         acApp.Quit
         Set acApp = Nothing
      End If
   End With
   GetReport = True
End Function
```

Opening a Secured Database through Automation

This function opens a secured Access database from Automation.

It takes the following arguments:

- strDbPath = full path to the secured database
- strUser = name of the user account to open the database
- strPassword = password of strUser
- strWkgrpPath = full path to the workgroup information file that contains strUser account

```
Function GetSecureDb(strDbPath As String, _
                     strUser As String, _
                     strPassword As String, _
                     strWrkgrpPath As String) As Boolean


   Dim acApp As Access.Application
   Dim strCommand As String

   Const APP_PATH As String = "c:\program files\microsoft
office\office\msaccess.exe"

   ' Build Access command line to pass to Shell function.
   strCommand = """" & APP_PATH & """" & " " & """" & strDbPath & """" _
                & " /User" & strUser & " /Pwd " & strPassword _
                & "/NoStartup" & " /Wrkgrp " & strWrkgrpPath

   ' Pass command line to Shell function. If this succeeds, pass
   ' the database path to the GetObject function and loop until
   ' the acApp object variable is initialized.
   If Shell(strCommand) Then
      Do
         On Error Resume Next
         Set acApp = GetObject(strDbPath)
            DoEvents
      Loop Until Err.Number = 0
   End If

   ' Sample command to open an Access form named "Categories".
   acApp.DoCmd.OpenForm "Categories"

   GetSecureDb = True

   ' Quit and destroy object variable.
   acApp.Quit
   Set acApp = Nothing
 End Function
```

Import a Text File into Access

The TransferText method is used to import a text file into Access. In this instance the database that we will import the file, d:\data\empTest.txt, into is d:\access2000\northwind.mdb. The parameter that follows acImportDelim is the File Specification. If the file is the default text file, that is, comma-separated, it is acceptable to omit the specification.

```
Sub newAccess()
   Dim objAccess As Access.Application
   Set objAccess = New Access.Application

   With objAccess
      .OpenCurrentDatabase "d:\access2000\northwind.mdb"
      .DoCmd.TransferText acImportDelim, , "Emp02", "d:\data\empTest.txt",
True
      .CloseCurrentDatabase
      .Quit
   End With
   Set objAccess = Nothing
End Sub
```

Sending the Current Record to Word.

The following example uses bookmarks in a Microsoft Word document to mark the locations where you want to place data from a record on a Microsoft Access form.

Creating a Microsoft Word Document

1. Start Microsoft Word and create the following new document:

First Last

Address City, Region, PostalCode

Dear Greeting,

Northwind Traders would like to thank you for your employment during the past year. Below you will find your photo. If this is not your most current picture, please let us know.

Photo

Sincerely,

Northwind Traders

2. Create a bookmark in Microsoft Word for the words "First," "Last," "Address," "City," "Region," "PostalCode," "Greeting," and "Photo":
    a. Select the word "First."
    b. On the Insert menu, click Bookmark
    c. In the Bookmark Name box, type "First," (without the quotation marks) and then click Add.
    d. Repeat steps 2a through 2c for each of the remaining words, substituting that word for the word "First" in steps 2a and 2c.
3. Save the document as C:\My Documents\MyMerge.doc, and then quit Microsoft Word.

Sending Data to Microsoft Word from a Microsoft Access Form

1. Start Microsoft Access and open the sample database Northwind.mdb.
2. Set a reference to the Microsoft Word 9.0 Object Library. To do so, follow these steps:
    a. Open any module in Design view.
    b. On the Tools menu, click References.
    c. Click Microsoft Word 9.0 Object Library in the Available References box. If that selection does not appear, browse for Msword9.olb, which installs by default in the C:\Program Files\Microsoft Office\Office folder.
    d. Click OK.
    e. Close the module.
3. Open the Employees form in Design view.
4. Add a command button to the form and set the following properties:

```
Command Button:
Name: MergeButton
Caption: Send to Word
OnClick: [Event Procedure]
```

5. Set the OnClick property of the command button to the following event procedure.

NOTE: In the following sample code, you must remove the comment from one line of code as indicated, depending on your version of Microsoft Access.

```
Private Sub MergeButton_Click()
   On Error GoTo MergeButton_Err
   Dim objWord As Word.Application
   ' Copy the Photo control on the Employees form.
   DoCmd.GoToControl "Photo"
   DoCmd.RunCommand acCmdCopy
   Start Microsoft Word.
   Set objWord = CreateObject("Word.Application")
   With objWord
```

```
      ' Make the application visible.
      .Visible = True
      ' Open the document.
      .Documents.Open ("c:\my documents\mymerge.doc")
      ' Move to each bookmark and insert text from the form.
      .ActiveDocument.Bookmarks("First").Select
      .Selection.Text = (CStr(Forms!Employees!FirstName))
      .ActiveDocument.Bookmarks("Last").Select
      .Selection.Text = (CStr(Forms!Employees!LastName))
      .ActiveDocument.Bookmarks("Address").Select
      .Selection.Text = (CStr(Forms!Employees!Address))
      .ActiveDocument.Bookmarks("City").Select
      .Selection.Text = (CStr(Forms!Employees!City))
      .ActiveDocument.Bookmarks("Region").Select
      .Selection.Text = (CStr(Forms!Employees!Region))
      .ActiveDocument.Bookmarks("PostalCode").Select
      .Selection.Text = (CStr(Forms!Employees!PostalCode))
      .ActiveDocument.Bookmarks("Greeting").Select
      .Selection.Text = (CStr(Forms!Employees!FirstName))
      ' Paste the photo.
      .ActiveDocument.Bookmarks("Photo").Select
      .Selection.Paste
   End With
   ' Print the document in the foreground so Word
   ' will not close until the document finishes printing.
   objWord.ActiveDocument.PrintOut Background:=False
   ' Close the document without saving changes.
   objWord.ActiveDocument.Close SaveChanges:=wdDoNotSaveChanges
   ' Quit Microsoft Word and release the object variable.
   objWord.Quit
   Set objWord = Nothing
   Exit Sub

MergeButton_Err:
   ' If a field on the form is empty
   ' remove the bookmark text and continue.
   If Err.Number = 94 Then
      objWord.Selection.Text = ""
      Resume Next
      ' If the Photo field is empty.
   ElseIf Err.Number = 2046 Then
      MsgBox "Please add a photo to this record and try again."
   Else
      MsgBox Err.Number & vbCr & Err.Description
   End If
   Exit Sub
End Sub
```

6. Save the Employees form and open it in Form view.
7. Click the Send To Word button to start Microsoft Word, merge data from the current record on the form into MyMerge.doc, print the document, and then close Microsoft Word.

NOTE: When you use this method of inserting text into a Word Document, you are deleting the bookmark when you insert the record field content. If you need to reference the text that you entered into the document, you mustbookmark it. You can use the following sample to add the bookmark "Last" to the text inserted from record field "LastName."

```
   .ActiveDocument.Bookmarks("Last").Select
   .Selection.Text = (CStr(Forms!Employees!LastName))
   ' add this line to reapply the bookmark name to the selection
   .ActiveDocument.Bookmarks.Add Name:="Last",Range:=Selection.Range
```

Using Automation with the Run-Time Version of Microsoft Access

To control a run-time installation of Access through Automation, there are special considerations to be aware of when the retail version of Microsoft Access is not installed:

1. To start a run-time instance of Microsoft Access (when one is not already running), you must use the Shell function and specify the path to Msaccess.exe and also a database to launch. This is because a run-time instance of Access cannot start without a database.
2. After starting the run-time instance, use the GetObject function to refer to the instance. (GetObject will only work with a run-time instance if the instance is already running.)
3. Only bring the instance into view when a database is open. If a database is not open in the run-time instance and you attempt to bring it into view, the instance will briefly display on the screen and then become minimized.
4. If you want to terminate a run-time instance, you must use the Quit method. For example, objAccess.Quit.

This procedure sets a module-level variable, objAccess, to refer to an instance of Access. The code first tries to use GetObject to refer to an instance that might already be open and contains the specified database (dbpath). If the database is not already open in an instance of Access, a new instance of the full version of Access is opened. If the full version of Access is not installed, the Shell function starts a run-time instance of Access. Once the instance is opened, you can use the CloseCurrentDatabase and OpenCurrentDatabase methods to work with other databases.

```
Sub OpenRunTime()
   Dim accpath As String, dbpath As String
   On Error Resume Next
   dbpath = "C:\Program Files\MyApp\MyApp.mdb"
   Set objAccess = GetObject(dbpath)
   If Err <> 0 Then
      If Dir(dbpath) = "" Then
         'dbpath is not valid
         MsgBox "Couldn't find database."
         Exit Sub
      Else 'The full version of Microsoft Access is not installed.
         accpath = "C:\Program Files\Microsoft Office\Art\Office\Msaccess.exe"
         If Dir(accpath) = "" Then
            MsgBox "Couldn't find Microsoft Access."
            Exit Sub
         Else Shell pathname:=accpath & " " & Chr(34) & dbpath &
Chr(34),windowstyle:=6
            Set objAccess = GetObject(dbpath)
         End If
      End If
   End If
End Sub
```

User Control and Visible Hints

When an application is launched by the user, the Visible and UserControl properties of the Application object are both set to True. When the UserControl property is set to True, it is not possible to set the Visible property of the Application object to False.

When an Application object is created using Automation, the Visible and UserControl properties of the Application object are both set to False.

If UserControl is True and an invalid command is passed to the instance of Access, Access will not suppress its own alert message and the instance of Access will not be terminated when its object variable is set to Nothing or loses scope.

Although you cannot directly change the UserControl property using code (it is read-only), Microsoft Access may change the UserControl property to True if the instance is brought into view using means other than setting the Application object's Visible property to True. However, if Visible was already True before the instance was brought into view, UserControl and Visible do not change. For example, suppose you have a public object variable, objAccess, that refers to an instance of Access. There is currently no code that is manipulating the instance and its UserControl and Visible properties are False. If the user then activates that instance of Access (by clicking on it from the TaskBar), its UserControl and Visible properties become True. The UserControl property of the instance will remain True until the user terminates it (by clicking File Exit, for example). After the user terminates the instance, it will become minimized instead of actually being terminated since objAccess still refers to it. Now, its UserControl and Visible properties become False again.

Understanding the UserControl property and whether the instance may have been created by the user is important because this affects how the instance is brought into view and how it is terminated. For example, when UserControl is True or the instance was created by the user, you cannot set its Visible property to bring the instance into view. Also, if you want to terminate an instance whose UserControl property is True, you must use the Quit method. For more information, see Using API calls to Bring Microsoft Access into View and Terminating the Instance of Access.

Using API calls to Bring Microsoft Access into View

To bring a new instance of Microsoft Access into view, you can set its Visible property to True. However, you cannot set the Visible property of an existing instance of Microsoft Access, or an instance of Microsoft Access whose UserControl property is True. The Visible property is read only when the UserControl property is True.

The UserControl property is True if the user started Microsoft Access manually, outside the context of your code. Even if the instance was created using Automation and its UserControl property is False, setting its Visible property to True will not bring the instance into view if its Visible property was already True and the instance was minimized by the user.

To reliably bring an existing instance into view, use the SetForegroundWindow or ShowWindow API calls. With the ShowAccess procedure, you pass it the object variable that refers to an instance of Access and it will bring the instance into view. It works whether the instance is new or existing and also allows an optional window "size" argument (e.g. to show the instance maximized)

```
Option Explicit

'The following Declare and Const statements are for the ShowAccess procedure.

Declare Function SetForegroundWindow Lib "User32" (ByVal hWnd As Long) As Long
Declare Function ShowWindow Lib "User32" (ByVal hWnd As Long, ByVal nCmdShow
As Long) As Long
Declare Function IsIconic Lib "User32" (ByVal hWnd As Long) As Long

Const SW_NORMAL = 1     'Show window in normal size
Const SW_MINIMIZE = 2   'Show window minimized
Const SW_MAXIMIZE = 3   'Show window maximized
Const SW_SHOW = 9       'Show window without changing window size

Dim objAccess As Access.Application

Sub ShowAccess(instance As Access.Application, Optional size As Variant)
    'Brings the instance of Access referred to by "instance" into view.
    'Size can be SW_NORMAL(1), SW_MINIMIZE(2), SW_MAXIMIZE(3), or

    'SW_SHOW(9).

    'If size is omitted, the size of the Access window is not changed
    '(SW_SHOW).
    'Calling example:
    '
    'ShowAccess instance:=objAccess, size:=SW_SHOW

    Dim hWnd As Long, temp As Long
    If IsMissing(size) Then size = SW_SHOW
    On Error Resume Next 'temporary error handler

    'Note: An error in the client occurs if you try to set the visible
    'of an instance of Access created by the user, regardless of its

    'UserControl setting. This is the reason for the temporary error
    'handling for the following line:

    If Not instance.UserControl Then instance.Visible = True
    On Error GoTo 0 'turn off error handler

    hWnd = instance.hWndAccessApp

    temp = SetForegroundWindow(hWnd)
```

```
    If size = SW_SHOW Then 'keep current window size
        If IsIconic(hWnd) Then temp = ShowWindow(hWnd, SW_SHOW)

    Else
        If IsIconic(hWnd) And size = SW_MAXIMIZE Then temp = _
            ShowWindow(hWnd, SW_NORMAL)
        temp = ShowWindow(hWnd, size)
    End If
End Sub

Sub StartAccess()
   Set objAccess = CreateObject("Access.Application")
   objAccess.OpenCurrentDatabase _
        ("C:\Program Files\Microsoft Office\Office\Samples\Northwind.mdb")
   ShowAccess objAccess, SW_MAXIMIZE
End Sub
```

Microsoft Binder

By default, the Microsoft Binder is marked 'Not Available' in Office 2000 Setup, and won't be installed on first use. To run the procedures in this section, re-run Setup, click Add or Remove Features, and then under Office Tools, select Microsoft Binder.

After installing Microsoft Binder, on the Tools menu use the References command to establish a reference to the 'Microsoft Binder 9.0 Object Library.'

Adding Documents to the Binder

This procedure demonstrates adding a Word document, a Worksheet, a Chart and a PowerPoint Show to a new Binder file using the **New** keyword. The error handler provides a clean exit from the procedure if the Binder file already exists.

```
Sub AddDocsToBinder()

   Dim bindApp As OfficeBinder.Binder

   On Error GoTo AddDocsToBinder_Err

   Set bindApp = New OfficeBinder.Binder

   With bindApp
      .Visible = True

      ' Add documents to binder.
      With .Sections
         .Add Type:="Word.Document"
         .Add Type:="Excel.Sheet"
         .Add Type:="Excel.Chart"
         .Add Type:="PowerPoint.Show"
      End With

      ' Save changes to the binder and close.
      .SaveAs "C:\My Documents\NewDocs.obd"
      .Close
   End With

 AddDocsToBinder_End:
   Set bindApp = Nothing
   Exit Sub

AddDocsToBinder_Err:

   Select Case Err.Number
      Case 6546
         MsgBox "File already exists."
      Case Else
         MsgBox "Error: " & Error.Number & " " & Error.Description
         Resume AddDocsToBinder_End
   End Select

End Sub
```

Delete Sections From a Binder

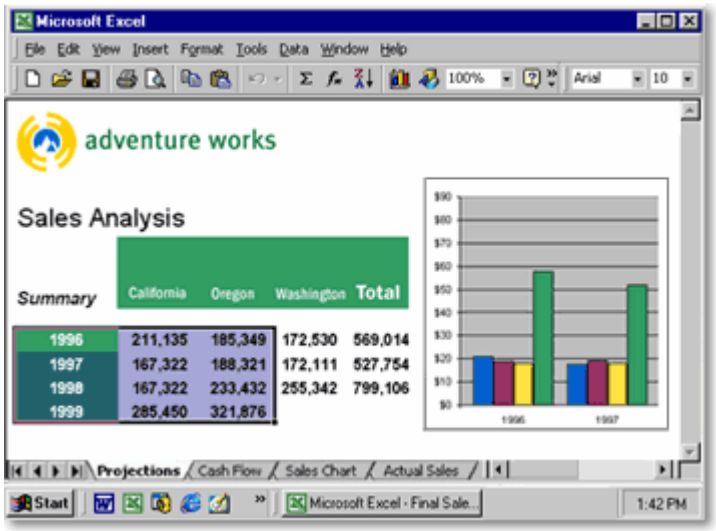
This procedure will delete all the documents from an existing binder.

```
Sub DeleteDocsFromBinder()

   Dim bindApp As OfficeBinder.Binder
   Dim intSectionCount As Integer

   Const ERR_FILE_EXISTS As Long = 6546

   On Error GoTo DeleteDocsFromBinder_Err

   ' Create new hidden instance of Binder.
   Set bindApp = New OfficeBinder.Binder

   With bindApp
      ' Make Binder visible and open binder created with
      ' AddDocsToBinder procedure.
      .Visible = True
      .Open ("C:\My Documents\NewDocs.obd")

      ' Loop through each section and delete it.
      For intSectionCount = .Sections.Count To 1 Step -1
         .Sections(intSectionCount).Delete
      Next
      .Save
      .Close
   End With

DeleteDocsFromBinder_End:
   Set bindApp = Nothing
   Exit Sub

DeleteDocsFromBinder_Err:
   Select Case Err.Number
      Case ERR_FILE_EXISTS
         MsgBox "File already exists."
      Case Else
         MsgBox "Error: " & Error.Number & " " & Error.Description
         Resume DeleteDocsFromBinder_End
   End Select
End Sub
```

Microsoft Excel

The Microsoft Office Spreadsheet

Discover better ways to analyze data and find solutions using Microsoft Excel 2000 and its streamlined spreadsheet creation tools, enhanced analysis tools, and powerful Web integration. Whether you are an expert or a novice, Excel will help you work more efficiently, turning your data into answers you can count on.



More Than Numbers

Excel 2000 provides comprehensive tools to help you create, analyze, and share spreadsheets. Create rich spreadsheets more easily than ever using enhanced formatting features. Analyze your data with charts, PivotTable® dynamic views, and graphs. And post your results to the Web for universal viewing and collaboration.

Microsoft Excel

The approach most commonly used to transfer data to an Excel workbook is Automation. Automation gives you the greatest flexibility for specifying the location of your data in the workbook as well as the ability to format the workbook and make various settings at run time. With Automation, you can use several approaches for transferring your data:

- Transfer data cell by cell
- Transfer data in an array to a range of cells
- Transfer data in an ADO recordset to a range of cells using the **CopyFromRecordset** method
- Create a **QueryTable** on an Excel worksheet that contains the result of a query on an ODBC or OLEDB data source
- Transfer data to the clipboard and then paste the clipboard contents into an Excel worksheet

There are also methods that you can use to transfer data to Excel that do not necessarily require Automation. If you are running an application server-side, this can be a good approach for taking the bulk of processing the data away from your clients. The following methods can be used to transfer your data without Automation:

- Transfer your data to a tab- or comma-delimited text file that Excel can later parse into cells on a worksheet
- Transfer your data to a worksheet using ADO

Differences From Earlier Versions of Microsoft Excel

The CreateObject and GetObject methods of Automation work differently when controlling Microsoft Excel 97 than they do when controlling earlier versions of Microsoft Excel. This is due to a design change in the Microsoft Excel object model. This topic explains the differences in behavior and offers some suggestions for making Automation code work with Microsoft Excel 97 and earlier versions of Microsoft Excel.

Behavior in Different Versions of Microsoft Excel

When you use CreateObject or GetObject in a macro to work with a Microsoft Excel sheet object, such as "Excel.Sheet" or "Excel.Sheet.8," the type of object the procedure returns is different for different versions of Microsoft Excel.

Version	Type of object returned
Microsoft Excel 97	Workbook
Microsoft Excel 5.0, 7.0	Worksheet

You can demonstrate the change in behavior by running the following Automation code from any Visual Basic for Applications client application such as Microsoft Access, Microsoft Word, or Microsoft Excel:

```
Sub ShowTypeName()

   Dim xlApp As Object

   Set xlAppj = CreateObject("Excel.Sheet")
   MsgBox TypeName(xlApp)
   xlApp.Quit
   Set xlApp = Nothing

End Sub
```

In Microsoft Excel 97, when you run the procedure, a message box that displays "Workbook" appears. In earlier versions of Microsoft Excel, the message is "Worksheet."

This change in behavior may cause a problem if your code uses properties and methods that are specific to the type of object to which the procedure references.

This following procedure works correctly with earlier versions of Microsoft Excel, because the Parent property of xlApp (a Worksheet object) is a Workbook object; and the Close method applies to workbooks:

```
Sub DemonstrateProblem()

   Dim xlApp As Object

   Set xlApp = CreateObject("Excel.Sheet")
   MsgBox TypeName(xlApp)
   xlApp.Parent.Close False
   Set xlApp = Nothing

End Sub
```

However, this procedure fails when you run it in Microsoft Excel 97, because the Parent property of xlApp (a Workbook object) is an Application object, and the Close method does not apply to the Application. When you run the procedure, you receive the following error message:

Run-time error '438':
Object doesn't support this property or method

Making your Code Work in All Versions of Microsoft Excel

If you want to use Automation with Microsoft Excel, but you do not know which version of Microsoft Excel is running, you can modify your code to work correctly with any version of Microsoft Excel.

One way to do this is to check the version of Microsoft Excel from the procedure, and then store the version number in a variable. To do this, use the following line of code:

```
ExcelVersion = Val(xlApp.Application.Version)
```

where "xlApp" is the name of the Microsoft Excel object.

The value of "ExcelVersion" is either 5, 7, or 8 for Microsoft Excel 5.0, 7.0, or 97 respectively.

After you determine the version of Microsoft Excel you are using, modify the procedure to work correctly with that version of Microsoft Excel. For example, you can make the procedure in this article work correctly by adding a few lines of code. The following example illustrates how to change the procedure:

```
Sub FixedProblem()

   Dim xlApp As Object
   ExcelVersion As Integer

   Set xlApp = CreateObject("Excel.Sheet")
   MsgBox TypeName(xlApp)
   ExcelVersion = Val(xlApp.Application.Version)
   If ExcelVersion = 8 Then       'For Microsoft Excel 97
      xlApp.Close False          'Close the workbook object.
   ElseIf ExcelVersion < 8 Then   'For Microsoft Excel 5.0 or 7.0
      xlApp.Parent.Close False   'Close the workbook object.
   End If
   Set xlApp = Nothing

End Sub
```

This procedure works correctly with Microsoft Excel 5.0, 7.0, or Microsoft Excel 97. The procedure also works correctly when you run it from Microsoft Visual Basic, Microsoft Word 97, or any other program (including Microsoft Excel) that supports Visual Basic or Visual Basic for Applications.

Determining if You Should Close the Instance of Microsoft Excel

One of the most important things to consider is if your code should close the instance of Microsoft Excel. The general rule is that if your code created the instance, it should also close the instance, unless you want the user to close it manually. If your code did not create the instance, then it should not close the instance; since it will affect other workbooks created by your users. If you want your code to always use an existing instance if possible, see Using a Pre-Existing Instance of Microsoft Excel.

Here's an example of how you could use the **IsExcelRunning()** function described in the above topic to determine if the code should close Microsoft Excel or not.

```
Sub SendDataToXL()
    Dim xlApp As Excel.Application
    Dim xlBook As Excel.Workbook
    Dim ExcelRunning As Boolean

    ExcelRunning = IsExcelRunning()
    If Not ExcelRunning Then
        Set xlApp = CreateObject("Excel.Application")
    Else
        Set xlApp = GetObject(, "Excel.Application")
    End If

    xlApp.Visible = True
    Set xlBook = xlApp.Workbooks.Add()
    xlBook.Worksheets(1).Cells(1, 1).Value = "Hello"
    xlBook.SaveAs "C:\Book1.xls"

    'If we started the instance, our code uses the
    'Quit method to close the instance
    If Not ExcelRunning Then xlApp.Quit
    Set xlBook = Nothing
    Set xlApp = Nothing

End Sub
```

Destroying an Instance of Microsoft Excel

When manipulating Automation servers, it is very important to destroy any object variables associated with the instances as well as to close the instance if your code created it. If not, your code could be opening instances of the server without closing them, and if run repeatedly, could eventually consume too many resources (such as memory) on the user's machine resulting in performance degradation. This topic will demonstrate how to correctly close an instance of Microsoft Excel and destroy any object variables associated with it. For more information about destroying object variables, see Destroying Automation Sessions.

When Will an Instance of Microsoft Excel Close Automatically During Automation?

An instance of Microsoft Excel will close automatically when its object variable loses scope, or is set to the keyword Nothing if there are no workbooks open, and the Application object's UserControl property is set to False.

How do I Close an Instance of Microsoft Excel?

There are two different methods for closing the instance of Microsoft Excel your code is automating. Regardless of which method you choose, your code should be responsible for destroying any object variables by setting them to the keyword Nothing. For information on whether your code should close the instance of Microsoft Excel, see Determining if You Should Close the Instance of Microsoft Excel.

Method 1: Use the Application Object's Quit Method

```
Option Explicit
Dim xlApp As Excel.Application

Sub CloseExcel()

   Set xlApp = CreateObject("Excel.Application")
   'Other Automation code here

   xlApp.Quit
   Set xlApp = Nothing
End Sub
```

Method 2: Closing All WorkBooks and Setting the UserControl Property to False

```
Option Explicit
Dim xlApp As Excel.Application

Sub CloseExcel()

   Set xlApp = CreateObject("Excel.Application")

   'Other Automation code here

   xlApp.WorkBooks.Close  'Close all open workbooks
   xlApp.UserControl = False
   Set xlApp = Nothing

End Sub
```

Using a Pre-Existing Instance of Microsoft Excel

To use a pre-existing instance of Microsoft Excel with Automation, use the GetObject function and specify "Excel.Application" as the Class type. If an instance of Microsoft Excel already exists, the GetObject function will return a reference to the instance. If an instance of Microsoft Excel does not already exist, your code will cause a trappable run-time error, and you can use the CreateObject function to create one.

You can use the following function to determine if an instance of Microsoft Excel is running.

```
Function IsExcelRunning() As Boolean
    Dim xlApp As Excel.Application
    On Error Resume Next
    Set xlApp = GetObject(, "Excel.Application")
    IsExcelRunning = (Err.Number = 0)
    Set xlApp = Nothing
    Err.Clear
End Function
```

And then, your code can determine whether it needs to create a new instance or not…

```
Sub ExcelInstance()
    Dim xlApp As Excel.Application

    Dim ExcelRunning As Boolean

    ExcelRunning = IsExcelRunning()
    If ExcelRunning Then
        Set xlApp = GetObject(, "Excel.Application")
    Else
        Set xlApp = CreateObject("Excel.Application")
    End If
    'Other automation code here...

    If Not ExcelRunning Then xlApp.Quit
    Set xlApp = Nothing
End Sub
```

Making an Instance of Microsoft Excel Visible

If a new instance of Microsoft Excel is created using Automation, it will be invisible by default. To display the newly created instance of Microsoft Excel, it is sometimes necessary to set the Visible property of the Application object to True. In previous versions, if you wished to use the Active property, you would need to make sure that the workbook's window was visible. With Office 2000 applications, it is necessary to set the Visible property to True if you intent to use any part of the server application's user interface.

For example, if you attempt to use Automation to print preview a range without first setting the Visible property of the workbook's window to True, the process may hang or produce a run-time error. The line, **xlApp.Visible = True**, allows Excel to display the Print Preview window.

```
Option Explicit

Sub printPreview()

   Dim xlApp as Excel.Application
   Set xlApp = CreateObject("Excel.Application")

   xlApp.Visible = True

  xlApp.Workbooks.Add
  xlApp.Sheets(1).Cells(1.1).Select
  xlApp.ActiveCell.Value = 10

  ' this line will cause the code to fail if Excel is not visible.
  xlApp.ActiveCell.PrintPreview

  xlApp.Quit
  Set xlApp = Nothing

End Sub
```

In this example, the code will run successfully without having to make Excel visible. In previous versions of Excel, if you wanted to use ActiveCell or ActiveSelection, it was necessary to set the visible property to True:

```
Option Explicit

Sub selectRange()

   Dim xlApp as Excel.Application
   Set xlApp = CreateObject("Excel.Application")

   xlApp.Workbooks.Add
   xlApp.Sheets(1).Cells(1.1).Select
   xlApp.ActiveCell.Value = 10

   xlApp.Quit
   Set xlApp = Nothing

End Sub
```

Opening a Microsoft Excel Workbook

This example demonstrates how to use the CreateObject function through Automation to open an existing Microsoft Excel workbook and display it to the user. Note that Microsoft Excel remains open even after the object variable has lost scope. See Destroying an Instance of Microsoft Excel for more information.

```
Sub OpenXLWorkBook(Path As String)

    Dim xlApp As Excel.Application

    'Check to see if the file name passed in to
    'the procedure is valid
    If Dir(Path) = "" Then
        MsgBox Path & " isn't a valid path!"
        Exit Sub
    Else
        Set xlApp = CreateObject("Excel.Application")

        'You do not need to make the application object visible
        'if you close the file and quit the application
        'later in your code in order to remove these objects
        'from memory.

        xlApp.Visible = True
        xlApp.Workbooks.Open Path
    End If

End Sub
```

This example demonstrates how to open an existing Microsoft Excel workbook through Automation using the GetObject function. Notice there are some differences in the code used with the GetObject function than with the CreateObject function. The primary difference is that we have to unhide the Window which contains the WorkBook, and set the WorkBook's Saved property to True to prevent Microsoft Excel from prompting the user to save changes upon exiting. See **Differences in the GetObject and CreateObject Functions** with Microsoft Excel for more information.

```
Sub OpenXLWorkBook(Path As String)

    Dim xlApp As Excel.Workbook
    Dim xlWindow As Excel.Window

    'Check to see if the file name passed in to
    'the procedure is valid
    If Dir(Path) = "" Then
        MsgBox "'" & Path & "' isn't a valid path!"
        Exit Sub
    Else
        Set xlApp = GetObject(Path)

        'Show the Excel Application Window
        xlApp.Parent.Visible = True

        'Unhide each window in the WorkBook
        For Each xlWindow In xlApp.Windows
            xlWindow.Visible = True
        Next

        'Prevent Excel from prompting to save changes
        'to the workbook when the user exits
        xlApp.Saved = True
    End If
```

```
End Sub
```

Note: Both of these examples are called from other procedures. You are not required to use the **Call** keyword when calling a procedure. However, if you use the **Call** keyword to call a procedure that requires arguments, argumentlist must be enclosed in parentheses. If you omit the **Call** keyword, you also must omit the parentheses around argumentlist. If you use either **Call** syntax to call any intrinsic or user-defined function, the function's return value is discarded. For example:

```
Sub Main()
   OpenXLWorkBook "c:\My Documents\test.xls"
End Sub

Sub Main()
   Call OpenXLWorkBook("c:\My Documents\test.xls")
End Sub
```

Adding a New Worksheet to an Existing Microsoft Excel Workbook

This example demonstrates how to use Automation to open an existing Microsoft Excel workbook and add a new worksheet to it. For information on opening Microsoft Excel workbooks through Automation, see Opening a Microsoft Excel Workbook**.** Note the use of the Quit method and the Nothing keyword to close the instance of Microsoft Excel and destroy its object variable. For information, see Destroying an Instance of Microsoft Excel.

```
Sub AddNewSheet()
    Dim xlApp As Excel.Application
    Set xlApp = CreateObject("Excel.Application")
    With xlApp
        .Workbooks.Open ("C:\Book1.XLS")
        .ActiveWorkbook.Sheets.Add
        .ActiveSheet.Name = "pivot"
        .ActiveWorkbook.Save
        .Quit
    End With
    Set xlApp = Nothing
End Sub
```

Using the Range Object With Microsoft Excel

To automate Microsoft Excel, you establish an object variable that usually refers to the Excel Application or Workbook object. Other object variables can then be set to refer to a Worksheet, a Range, or other objects in the Microsoft Excel object model. When you write code to use an Excel object, method, or property, you should always precede the call with the appropriate object variable. If you do not, Visual Basic establishes its own reference to Excel. This reference might cause problems when trying to run the automation code multiple times. Note that even if the line of code begins with the object variable, there may be a call to an Excel object, method, or property in the middle of the line of code that is not preceded with an object variable.

This example demonstrates how to use Automation to refer to a Range in Microsoft Excel, and then set the value of it. Note the following line of code from the sample:

- xlSheet.Range(xlSheet.Cells(1,1), xlSheet.Cells(3,3)).Value = 1000

Common practise referencing a range within Excel may lead you to enter the code as:

- xlSheet.Range("A1:C3").Value = 1000

This syntax causes Visual Basic to establish a reference to Excel because the part of the Range object has not been qualified with an Excel object variable, in this case, xlSheet. Visual Basic does not release this reference until you end the program.

To use this example, create a workbook named Book1.xls in C:\My Documents. Note that the instance of Excel will not be visible to the user.

```
Sub usingRange()

   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   Dim xlSheet As Excel.Worksheet

   Set xlApp = CreateObject("Excel.Application")
   Set xlBook = xlApp.Workbooks.Open(filename:="c:\My Documents\Book1.xls")

   'Set reference to Worksheet object
   Set xlSheet = xlBook.Sheets("Sheet3")

   'Puts 1000 in cells A1 through C3 of Sheet3 in Book1.xls
   xlSheet.Range(xlSheet.Cells(1,1), xlSheet.Cells(3,3)).Value = 1000

   xlBook.Close savechanges:=True

   'Close Microsoft Excel and destroy object variables
   xlApp.Quit
   Set xlSheet = Nothing
   Set xlBook = Nothing
   Set xlApp = Nothing

End Sub
```

Adding a Named Range to a Workbook

This example demonstrates how to use Automation to open a Microsoft Excel workbook and determine the address of the current region. The current region is a range bounded by any combination of blank rows and blank columns. The subroutine then gives that range a name. This procedure assumes you have already created a new Microsoft Excel workbook which contains data.

```
Sub SetRange(Path As String)
    Dim xlApp As Excel.Application
    Dim rng As String

    'Check to see if the file name passed in to
    'the procedure is valid
    If Dir(Path) = "" Then
        MsgBox Path & " isn't a valid path!"
        Exit Sub
    Else
        Set xlApp = CreateObject("Excel.Application")
        xlApp.Visible = True
        xlApp.Workbooks.Open Path
        xlApp.ActiveSheet.Range("a1").Select
        rng = xlApp.Selection.CurrentRegion.Address
        xlApp.ActiveWorkbook.Names.Add "DataRng", "=sheet1!" & rng
    End If
End Sub
```

Selecting a Specific Location on a Microsoft Excel Worksheet

This example demonstrates how to use Automation to select a specific range of cells in a Microsoft Excel worksheet. This example uses the Range method to select the desired cells. See Using the Range Object With Microsoft Excel for more information.

```
Sub SelectCells()

   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   Dim xlSheet As Excel.Worksheet

   Set xlApp = CreateObject("Excel.Application")
   Set xlBook = xlApp.Workbooks.Open(FileName:="c:\My Documents\Book1.xls")
   Set xlSheet = xlBook.ActiveSheet
   xlApp.Visible = True

   With xlSheet
     ' insert range selection example code here
   End With

End Sub
```

Selects cells B1:F7, then makes cell D2 the active cell.

```
   .Range(xlSheet.Cells(1, 2), xlSheet.Cells(7, 6)).Select
   .Range("D2").Activate
```

Selects an area named "MyArea"

```
   .Range("MyArea").Select
```

Selects the last cell in the used range on the ActiveSheet

```
   .Cells.SpecialCells(xlCellTypeLastCell).Select
```

Selects all the cells that contain formula.

```
   .Cells.SpecialCells(xlCellTypeFormulas).Select
```

For more information on the SpecialCells method, search Microsoft Visual Basic Help for SpecialCells.

Editing a Microsoft Excel Workbook

When using Automation to edit a Microsoft Excel workbook, keep the following in mind.

Creating a new instance of Microsoft Excel and opening a workbook results in an invisible instance of Microsoft Excel, **and a hidden instance of the workbook**. Thus, if you edit the workbook and save it, the workbook is saved hidden. The next time the user opens Microsoft Excel manually, the workbook will be invisible and the user has to select Unhide from the Window menu to actually view the workbook.

To prevent this, your Automation code should unhide the workbook before editing it and saving it. Note that this does NOT mean Microsoft Excel itself has to be visible, so the user will never actually see it.

Consider the following example.

```
Sub usingRange()

   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   Dim xlSheet As Excel.Worksheet

   ' set reference to Application object
   Set xlApp = CreateObject("Excel.Application")

   ' set reference to Workbook object
   Set xlBook = xlApp.Workbooks.Open("C:\BOOK1.XLS")

   ' set the reference to Worksheet object
   Set xlSheet = xlBook.WorkSheet(1)

   ' puts 1000 in cells A1 through C3 of Sheet1 in Book1.xls
   xlSheet.Range(xlSheet.Cells(1,1), xlSheet.Cells(3,3).Value = 1000
   xlBook.Close savechanges:=True

   ' close Excel and destroy object variables
   xlApp.Quit
   Set xlSheet = Nothing
   Set xlBook = Nothing
   Set xlApp = Nothing

End Sub
```

The above example opens an existing workbook in a new, hidden instance of Microsoft Excel, edits the workbook, and then saves and closes it. Since the workbook was opened in an invisible instance of Microsoft Excel and was never unhidden, the code has now saved the workbook as a hidden workbook. To avoid this problem, modify the example to unhide the workbook before editing it as shown below.

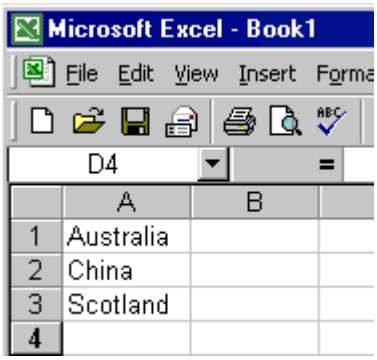
```
Sub usingRange()

   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   Dim xlSheet As Excel.Worksheet

   ' set reference to Application object
   Set xlApp = CreateObject("Excel.Application")

   ' set reference to Workbook object
   Set xlBook = xlApp.Workbooks.Open("C:\BOOK1.XLS")

   ' set the reference to Worksheet object
   Set xlSheet = xlBook.WorkSheet(1)

   ' unhide the workbook.  Note that this does NOT
```

```
    ' unhide the instance of Microsoft Excel
    xlApp.Windows(1).Visible = True

    ' puts 1000 in cells A1 through C3 of Sheet1 in Book1.xls
    xlSheet.Range(xlSheet.Cells(1,1), xlSheet.Cells(3,3).Value = 1000
    xlBook.Close savechanges:=True

    ' close Excel and destroy object variables
    xlApp.Quit
    Set xlSheet = Nothing
    Set xlBook = Nothing
    Set xlApp = Nothing

End Sub
```

Populating a List Box with Data From Excel.

The following function uses Automation to retrieve a list of countries from a Microsoft Excel worksheet. The list is then used to populate a list box bound to the Country field of the Suppliers table in the sample database Northwind.mdb.

1. Start Microsoft Excel and create a new worksheet with the following data:



2. Save the worksheet as C:\My Documents\Country.xls.
   NOTE: If you change the name or location of this file, be sure to change the sample code to reflect this change.
3. Open the sample database Northwind.mdb and create a new module.
4. Type the following lines in the Declarations section:

```
Option Explicit
Dim Countries(3) As String
```

5. Type the following subroutine:

```
Sub OLEFillCountries()
   Dim i%
   Dim XL As Object
   Dim WrkBook As Object
   Set XL = CreateObject("Excel.Application")
   Set WrkBook = XL.Workbooks.Open("C:\My Documents\Country.xls")
   For i% = 0 To 2
      Countries(i%) = WrkBook.Sheets(1).Cells(i% + 1, 1).Value
   Next i%
   XL.Quit
   Set WrkBook = Nothing
   Set XL = Nothing
End Sub
```

6. Type the following function:

```
Function OLEFillList(fld As Control, id, row, col, code)
   Select Case code
      Case 0 ' Initialize.
         Call OLEFillCountries
         OLEFillList = True
      Case 1 ' Open.
         OLEFillList = id
      Case 3 ' Get number of rows.
         OLEFillList = 3
      Case 4 ' Get number of columns.
         OLEFillList = 1
      Case 5 ' Force default width.
         OLEFillList = -1
      Case 6
         OLEFillList = Countries(row)
   End Select
End Function
```

7. Save the module as "OLE Fill list box" (without the quotation marks).
8. Create a new form based on the Suppliers table.
9. Create a list box with the following properties:

```
Object: List Box
ControlSource: Country
RowSourceType: OLEFillList
```

10. Open the form in Form view.

    Note that the list box contains the values entered in the spreadsheet.

Opening a CSV file with Microsoft Excel - from Microsoft Access.

This example demonstrates opening and formating a Comma Separated Values (CSV) file using Microsoft Access as the Automation Client and Microsoft Excel as the Automation Server.

```
Sub ProcessFile()

   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   Dim xlSheet As Excel.Worksheet
   Dim xlSelection As Range
   Dim rowCounter As Integer
   Dim varTemp As Variant

   Const CSVFILE = "C:\My Documents\CSVTest.csv"
   Const NEWFILE = "C:\My Documents\Test.xls"

   Set xlApp = CreateObject("Excel.Application")
   xlApp.Visible = True
   Set xlBook = xlApp.Workbooks.Open(filename:=CSVFILE)

   Set xlSheet = xlBook.ActiveSheet
   Set xlSelection = xlSheet.Range(xlSheet.Cells(1, 1), xlSheet.Cells(1,
1).End(xlToRight))

   With xlSelection
      With .Interior
         .ColorIndex = 15
         .Pattern = xlSolid
      End With
   End With

   Set xlSelection = xlSheet.Range("A1").CurrentRegion

   With xlSelection
      With .Borders
         .LineStyle = xlContinuous
         .Weight = xlThin
      End With
      .Columns.AutoFit
   End With

   rowCounter = 2
   Set xlSelection = xlSheet.Cells(rowCounter, 1)
   varTemp = xlSelection.Value
   rowCounter = rowCounter + 1
   Set xlSelection = xlSheet.Cells(rowCounter, 1)

   Do While xlSelection.Value <> ""
      If xlSelection.Value = varTemp Then
         xlSelection.Value = ""
         xlSelection.Borders(xlEdgeTop).LineStyle = xlLineStyleNone
      Else
         varTemp = xlSelection.Value
      End If
      rowCounter = rowCounter + 1
      Set xlSelection = xlSheet.Cells(rowCounter, 1)
   Loop

   On Error Resume Next
```

```
    Kill NEWFILE
    On Error Goto 0
    xlBook.SaveAs filename:=NEWFILE, FileFormat:=xlWorkbookNormal

    xlBook.Close
    Set xlSheet = Nothing
    Set xlBook = Nothing
    xlApp.Quit
    Set xlApp = Nothing

End Sub
```

Copy Formulas

This example opens a workbook and, after selecting all the formulas on the worksheet, copies all of the worksheet formulas from the ActiveSheet to Sheet 2.

```
Sub CopyFormulas()
   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   Dim xlSheet As Excel.Worksheet
   Dim rangeItem as Range
   Set xlApp = CreateObject("Excel.Application")
   Set xlBook = xlApp.Workbooks.Open(FileName:="c:\My Documents\Book1.xls")
   Set xlSheet = xlBook.ActiveSheet
   xlApp.Visible = True
   With xlSheet
      .Cells.SpecialCells(xlCellTypeFormulas).Select
      For Each rangeItem in Selection
         ' copy formula from the active sheet to Sheet2
          xlBook.Worksheets("Sheet2").Range(rangeItem.Address) =
rangeItem.Formula
        Next rangeItem
   End With
End Sub
```

Use Automation to Transfer an Array of Data to a Range on a Worksheet:

An array of data can be transferred to a range of multiple cells at once:

```
Sub bulkTransfer()

   Dim xlApp As Excel.Application
   Dim xlBook As Workbook
   Dim xlSheet As Worksheet

   'Start a new workbook in Excel
   Set xlApp = CreateObject("Excel.Application")
   Set xlBook = xlApp.Workbooks.Add

   'Create an array with 3 columns and 100 rows
   Dim aryData(1 To 100, 1 To 3) As Variant
   Dim intCount As Integer

   For intCount = 1 To 100
      aryData(intCount, 1) = "ORD" & Format(r, "0000")
      aryData(intCount, 2) = Rnd() * 1000
      aryData(intCount, 3) = aryData(intCount, 2) * 0.7
   Next

   'Add headers to the worksheet on row 1
   Set xlSheet = xlBook.Worksheets(1)
   xlSheet.Range(xlSheet.Cells(1,1),xlSheet.Cells(1,3)).Value = Array("Order
ID", "Amount", "Tax")

   'Transfer the array to the worksheet starting at cell A2
   xlSheet.Range("A2").Resize(100, 3).Value = aryData

   'Save the Workbook and Quit Excel
   xlBook.SaveAs "C:\My Documents\ArrayDump.xls"
   xlApp.Quit
   Set xlSheet = Nothing
   Set xlBook = Nothing
   Set xlApp = Nothing

End Sub
```

If you transfer your data using an array rather than cell by cell, you can realize an enormous performance gain with a large amount of data. Consider this line from the code above that transfers data to 300 cells in the worksheet:

```
      xlSheet.Range("A2").Resize(100, 3).Value = aryDatay
```

This line represents two interface requests (one for the Range object that the Range method returns and another for the Range object that the Resize method returns). On the other hand, transferring the data cell by cell would require requests for 300 interfaces to Range objects. Whenever possible, you can benefit from transferring your data in bulk and reducing the number of interface requests you make.

Use Automation to Transfer an ADO Recordset to a Worksheet Range

Excel 2000 provides a CopyFromRecordset method that allows you to transfer an ADO (or DAO) recordset to a range on a worksheet. The following code illustrates how you could automate Excel 2000 and transfer the contents of the Orders table in the Northwind Sample Database using the CopyFromRecordset method.

Remember to set the reference to the current Microsoft ActiveX Data Objects Library.

```
Sub transferRecordset()

   'Create a Recordset from all the records in the Orders table
   Dim sNWind As String
   Dim conn As New ADODB.Connection
   Dim rs As ADODB.Recordset

   sNWind = "C:\Program Files\Microsoft Office\Office\Samples\Northwind.mdb"
   conn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & sNWind & ";"
   conn.CursorLocation = adUseClient
   Set rs = conn.Execute("Orders", , adCmdTable)

   'Create a new workbook in Excel
   Dim xlApp As Object
   Dim xlBook As Object
   Dim xlSheet As Object

   Set xlApp = CreateObject("Excel.Application")
   Set xlBook = xlApp.Workbooks.Add
   Set xlSheet = xlBook.Worksheets(1)

   'Transfer the data to Excel
   xlSheet.Range("A1").CopyFromRecordset rs

   'Save the Workbook and Quit Excel
   xlBook.SaveAs "c:\My Documents\ADOExample.xls"
   xlApp.Quit

   'Close the connection
   rs.Close
   conn.Close

End Sub
```

For more information on ADO see Microsoft ActiveX Data Objects. For more information on DAO see Microsoft Data Access Objects.

Changing a Graph's Data Marker's Shape

Using Automation, it is possible to change the shapes of data markers on a Microsoft Graph chart. You can emphasize individual data points with various graphical marks. You can use the MarkerStyle property to change the appearance of this graphical mark.

NOTE: The MarkerStyle property only applies to graphs of type "Line" and "Scatter". If the foreground and background colors are the same, then some of the shapes above will appear as filled squares. If the MarkerStyle property is set to Automatic (-4105) then the colors will be reset also.

```
Function SetMarkerStyle(GraphObj As Object)

   Dim SeriesCount As Integer

   With GraphObj
      ' loop through each data series and change the marker
      ' style to a circle.  For a list of other constants
      ' to use with the MarkerStyle property, view the
      ' Microsoft Graph object library in the Object Browser
      For SeriesCount = 1 To .SeriesCollection.Count
         .SeriesCollection(sCount).MarkerStyle = xlMarkerStyleCircle
      Next
    End With

End Function
```

Changing a Microsoft Graph Chart Type

Using Automation, it is possible to set or retrieve the type of a Microsoft Graph chart. To do this, you must set the Type property of the Graph object. Be aware that changing a graph from one type to another automatically resets properties of the graph that do not apply to the new graph type. For example, trend lines apply to a two dimensional Column graph but do not apply to a Pie chart. Therefore, changing the type from a Column to a Pie will drop the trend lines.

```
Function SetChartType(GraphObj As Object)

    ' set Chart type to three-dimensional column.
    ' for a list of other constants to use with the
    ' ChartType property, view the Microsoft Graph object
    ' library in the Object Browser
    GraphObj.ChartType = xl3DColumn

End Function
```

Changing the Border Properties of a Graph

Using Automation, it is possible to change the border weight, color, and linestyle of a Microsoft Graph chart. Use the Weight property to set the thickness of the border, the Color property to change the border's color, or the LineStyle property to change the border styles (i.e. a solid line, dashed line, etc...).

The following example sets the border line style to a dashed dot line, the border weight to thick, and the border color to red:

```
Function SetBorderStyle(GraphObj As Object)

    'For a list of other constants to use with these
    'properties, select the Microsoft Graph object
    'library in the Object Browser and search on the
    'property name

    With GraphObj.ChartArea.Border
        .LineStyle = xlDashDot
        .Weight = xlThick
        .Color = 255
    End With

End Function
```

Note: To remove a border from a graph, set the LineStyle property to xlLineStyleNone.

Changing the Line Texture on a Graph

It is possible through Automation to change the texture of lines on a Microsoft Graph chart. Some graphs contain lines contecting the data points being graphed. These lines can appear jagged. Use the Smooth property to remove some of the jagged edges. Note that the Smooth property applies to Line and Scatter graphs only.

The following example toggles the first series between smoothed and normal lines:

```
Function ToggleSmoothProperty(GraphObj As Object)

    Dim SeriesCount As Integer

    With GraphObj

        ' loop through each data series in the chart
        'and toggle the Smooth property of the Series

        For SeriesCount = 1 To .SeriesCollection.Count
            .SeriesCollection(SeriesCount).Smooth = _
            Not .SeriesCollection(SeriesCount).Smooth
        Next

    End With

End Function
```

Changing the Range of an Axis

It is possible through Automation to change the range of an axis on a Microsoft Graph chart. Some graphs contain an X or Y axis that display a scale by which the data points can be measured. These scales usually begin at the value 0 and extend to a value sufficient to measure the largest point being graphed.

The MinimumScale and MaximumScale properties allow a graph to alter the range of this scale. This property applies only to the "value" axis which is axis "2".

```
Function SetMinMaxScale(GraphObj As Object)

    With GraphObj

        .Axes(2).MinimumScale = 30
        .Axes(2).MaximumScale = 90

    End With

End Function
```

Manipulating a Graph's Legend

With Automation, it is possible to show, hide, and position the legend of a Microsoft Graph chart. The HasLegend property can be used to determine if the graph is currently displaying a legend as well as changing the legend's visibility. The Position property can be used to place the legend in various locations.

The Position property only sets the location of a graph's legend. Use the HasLegend property

to ensure the legend is visible. Attempting to change the Position property while the HasLegend property is set to False will result in a run-time error. The graph size may shrink to accommodate the change in the legend's position.

The following example demonstrates how to create a legend with a red border, place it at the bottom of the graph, set the Font name and size, and add a Shadow to it.

```
Function SetLegend(GraphObj As Object)

    With GraphObj

        'Create a legend
        .HasLegend = True

        'Set the position of the legend.  For other
        'constants you can use with the Legend property,
        'view Graph object model in the Object Browser.
        .Legend.Position = xlLegendPositionBottom
        .Legend.Font.Name = "Arial"
        .Legend.Font.Size = 14
        .Legend.Shadow = True

    End With

End Function
```

The following example demonstrates how to toggle the graph legend on and off:

```
Function ToggleLegend(GraphObj As Object)

    GraphObj.HasLegend = Not GraphObj.HasLegend

End Function
```

Microsoft Outlook

The Microsoft Office E-mail and Personal Information Manager

Information management is a vital task for computer users today who must juggle everything from electronic mail and calendars to contacts and task lists. The Microsoft Outlook® 2000 messaging and collaboration client helps you organize all this information and improve communication and collaboration across your enterprise. And because it works like the rest of Microsoft Office, Outlook 2000 is easy to learn and use.

One Window to Your World of Information

helps you organize and manage all your information from a single location. And in today's workplace, more efficient information management means increased productivity—and better bottom-line results.

Printing Messages From Outlook.

This example demonstrates how to loop through all messages in the Inbox and print each one.

```
Sub printMessages()

   Dim olookApp As Outlook.Application
   Dim olookMsg As Object
   Dim olookSpace As Outlook.NameSpace
   Dim olookFolder As Outlook.MAPIFolder

   Set olookApp = CreateObject("Outlook.Application")
   Set olookSpace = olookApp.GetNameSpace("MAPI")
   Set olookFolder = olookSpace.GetDefaultFolder(olFolderInbox)

   ' loop through each message in the Inbox and print it.
   ' each message will be printed in a separate print job.
   For Each olookMsg In olookFolder.Items
      olookMsg.PrintOut
   Next

   Set olookFolder = Nothing
   Set olookSpace = Nothing

   ' quitting Outlook will close the instance that the user may
   ' currently be using.
   olookApp.Quit
   Set olookApp = Nothing

End Sub
```

Creating a New Folder in Microsoft Outlook

This example demonstrates how to create a new Microsoft Outlook folder using Automation. To create a new folder, you must first reference the folder object you want to create the folder in. In this example, we use the GetDefaultFolder method of the NameSpace object to refer to the Inbox folder. However, since we do not want to create the new folder as a subfolder under the Inbox, we must use the Parent property to refer to the parent of the Inbox. Thus, the new folder gets created on the same level as the Inbox.

```
Sub addFolder()

    Dim olookApp As Outlook.Application
    Dim olookSpace As Outlook.NameSpace
    Dim olookInbox As Outlook.MAPIFolder
    Dim olookFolder As Outlook.MAPIFolder

    Set olookApp = CreateObject("Outlook.Application")
    Set olookSpace = olookApp.GetNamespace("MAPI")

    'Must reference the folder we wish to create the new folder in.
    Set olookInbox = olookSpace.GetDefaultFolder(olFolderInbox).Parent

    'Use the Add method of the Folders collection of the MAPIFolder
    'object returned in the above statement.
    Set olookFolder = olookInbox.Folders.Add("MyNewFolder")

    Set olookFolder = Nothing
    Set olookInbox = Nothing
    Set olookSpace = Nothing
    Set olookApp = Nothing

End Sub
```

Sending an Outlook Message With an Attachment.

There are six main steps to sending a Microsoft Outlook mail message by using Automation, as follows:

1. Initialize the Outlook session.
2. Create a new message.
3. Add the recipients (To, CC, and BCC) and resolve their names.
4. Set valid properties, such as the Subject, Body, and Importance.
5. Add attachments (if any).
6. Send the message.

To send a Microsoft Outlook mail message programmatically, follow these steps:

1. Create a sample text file named Customers.txt in the C:\My Documents folder.
2. Launch an Office application and open the Visual Basic Editor.
3. Create a module and type the following line in the Declarations section if it is not already there:

```
Option Explicit
```

4. On the Tools menu, click References.
5. In the References box, click to select the Microsoft Outlook 9.0 Object Library, and then click OK.

   NOTE: If the Microsoft Outlook 9.0 Object Library does not appear in the Available References box, browse your hard disk for the file, Msoutl9.olb. If you cannot locate this file, you must run the Microsoft Outlook Setup program to install it before you proceed with this example.

6. Type the following procedure in the new module:

```
Sub sendMessage(Optional AttachmentPath)

    Dim olookApp As Outlook.Application
    Dim olookMsg As Outlook.MailItem
    Dim olookRecipient As Outlook.Recipient
    Dim olookAttach As Outlook.Attachment

    ' create the Outlook session.
    Set olookApp = CreateObject("Outlook.Application")

    ' create the message.
    Set olookMsg = olookApp.CreateItem(olMailItem)

    With olookMsg
       ' add the To recipient(s) to the message.
       Set olookRecipient = .Recipients.Add("Christopher Wyke")
       olookRecipient.Type = olTo

       ' add the CC recipient(s) to the message.
       Set olookRecipient = .Recipients.Add("Robert Dil")
       olookRecipient.Type = olCC

       ' set the Subject, Body, and Importance of the message.
       .Subject = "This is an Automation test with Microsoft Outlook"
       .Body = "Last test - I promise." & vbCrLf & vbCrLf
       .Importance = olImportanceHigh  'High importance

       ' add attachments to the message.
       If Not IsMissing(AttachmentPath) Then
          Set olookAttach = .Attachments.Add(AttachmentPath)
       End If

       ' resolve each Recipient's name
       For Each olookRecipient In .Recipients
          olookRecipient.Resolve
          If Not olookRecipient.Resolve Then
             olookMsg.Display   ' display any names that can't be
```

```
resolved
45.          End If
46.       Next
47.          .Send
48.
49.       End With
50.       Set olookMsg = Nothing
51.       Set olookApp = Nothing
52.
53.    End Sub
```

54. To test this procedure, type the following line in the Immediate window, and then press ENTER:

```
SendMessage "C:\My Documents\Customers.txt"
```

To send the message without specifying an attachment, omit the argument when calling the procedure, as follows:

```
SendMessage
```

Adding Notes to Microsoft Outlook

This example demonstrates how to add a new note to Microsoft Outlook using Automation.

```
Sub addNote()

    Dim olookApp As Outlook.Application
    Dim olookNote As Outlook.NoteItem

    Set olookApp = CreateObject("Outlook.Application")
    Set olookNote = olookApp.CreateItem(olNoteItem)

    With olookNote
        .Body = "Body of my note."

        ' set the color of the note.
        ' Can be one of the following OlNoteColor constants:
        ' olBlue(0), olGreen(1), olPink(2), olWhite(4), or olYellow(3).

        .Color = olBlue
        .Save
        .Display
    End With

    Set olookNote = Nothing
    Set olookApp = Nothing

End Sub
```

Adding Tasks to Microsoft Outlook

This example demonstrates how to add new tasks to Microsoft Outlook using Automation.

```
Sub addTask()

    Dim olookApp As Outlook.Application
    Dim olookTask As Outlook.TaskItem

    Set olookApp = CreateObject("Outlook.Application")
    Set olookTask = olookApp.CreateItem(olTaskItem)

    With olookTask
        .Subject = "This is the subject of my task"
        .Body = "This is the body of my task."
        .ReminderSet = True

        'Set to remind us 2 minutes from now.
        .ReminderTime = DateAdd("n", 2, Now)

        'Set the due date to 5 minutes from now.
        .DueDate = DateAdd("n", 5, Now)
        .ReminderPlaySound = True

        'Add the path to a .wav file on your computer.
        .ReminderSoundFile = "C:\Windows\Media\Ding.WAV"
        .Save
    End With

    Set olookTask = Nothing
    Set olookApp = Nothing

End Sub
```

Starting a Session of Microsoft Outlook With a Different Profile

Normally, instantiating a session of Microsoft Outlook will cause you to use the default Outlook profile which is located under the following key in the Windows registry:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Windows Messaging Subsystem\Profiles
```

However, sometimes it is useful to create an instance of Microsoft Outlook using a different profile. This example demonstrates how to use a different profile when creating a new instance of Microsoft Outlook through Automation.

```
Sub StartOutLook(ProfileName As String)

   Dim olookApp As Outlook.Application
   Dim olookSpace As Outlook.NameSpace

   Set olookApp = CreateObject("Outlook.Application")
   Set olookSpace = olookApp.GetNamespace("MAPI")
   olookSpace.Logon ProfileName

   ' use the Display method to actually show the Outlook
   ' session and view the Inbox folder.
   olookSpace.GetDefaultFolder(olFolderInbox).Display>

   Set olookSpace = Nothing
   Set olookApp = Nothing

End Sub
```

Import Outlook Items from an Access Database

This example provides the sample code for creating Microsoft Outlook contacts from information stored in a Microsoft Access database. Establish a reference to the Microsoft Outlook 9.0 Object Library and to the Microsoft DAO 3.6 Object Library.

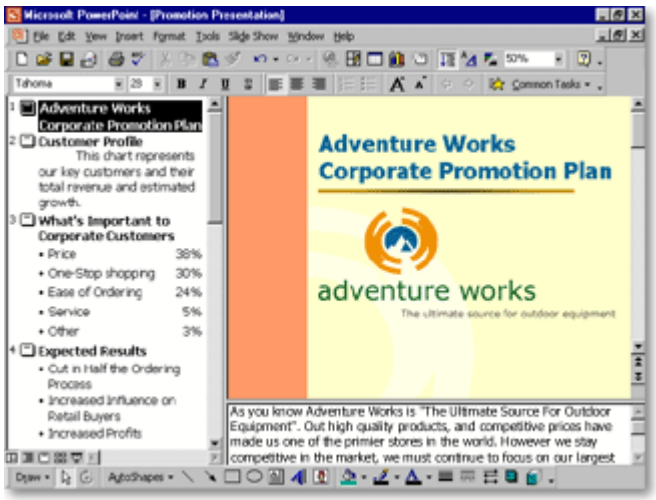
```
Sub exportAccessContactsToOutlook()

   Const DBLOCATION = "olookContact:\Program Files\Microsoft Office" _
      & "\Office\Samples\Northwind.mdb"

   ' set up DAO Objects.
   Dim db As DAO.Database
   Dim rst As DAO.Recordset
   Set db = OpenDatabase(DBLOCATION)
   Set rst = db.OpenRecordset("Customers")

   ' set up Outlook Objects.
   Dim olookApp As New Outlook.Application
   Dim olookSpace As Outlook.NameSpace
   Dim olookFolder As Outlook.MAPIFolder
   Dim olookContact As Outlook.ContactItem
   Dim olookUserProp As Outlook.UserProperty

   Set olookSpace = olookApp.GetNamespace("MAPI")
   Set olookFolder = olookSpace.GetDefaultFolder(olFolderContacts)

   With rst
      .MoveFirst

      ' loop through the Microsoft Access records.
      Do While Not .EOF

         ' create a new Contact item.
         Set olookContact = olookApp.CreateItem(olContactItem)

         ' specify which Outlook form to use.
         ' change "IPM.Contact" to "IPM.Contact.<formname>" if you've
         ' created a custom Contact form in Outlook.
         olookContact.MessageClass = "IPM.Contact"

         ' create all built-in Outlook fields.
         If ![CompanyName] <> "" Then olookContact.CompanyName =
![CompanyName]
         If ![ContactName] <> "" Then olookContact.FullName = ![ContactName]

         ' create the first user property (UserField1).
         Set olookUserProp = olookContact.UserProperties.Add("UserField1",
olText)

         ' Set its value.
         If ![CustomerID] <> "" Then olookUserProp = ![CustomerID]

         ' create the second user property (UserField2).
         Set olookUserProp = olookContact.UserProperties.Add("UserField2",
olText)

         ' set its value and so on....
         If ![Region] <> "" Then olookUserProp = ![Region]

         ' save the contact.
```

```
            olookContact.Save

            .MoveNext
         Loop
      End With

      ' clean up
      Set olookUserProp = Nothing
      Set olookContact = notheing
      Set olookFolder = Nothing
      Set olookSpace = Nothing
      Set olookApp = Nothing

      Set rs = Nothing
      db.Close
      Set db = Nothing

End Sub
```

Microsoft PowerPoint

The Microsoft Office Presentation Graphics Program

Microsoft PowerPoint® 2000 makes it easier to organize, illustrate, and deliver your ideas professionally. Whether you're conducting a meeting, presenting at a conference, or delivering your message over the Internet, the Microsoft PowerPoint 2000 presentation graphics program provides the tools you need to make your point—powerfully.



It's Easier to Make Your Point—Anywhere

Organize and illustrate your ideas faster using the professional design techniques and familiar Microsoft Office tools in PowerPoint 2000. And use real-time Internet technology to collaborate and communicate with impact to a wider audience than ever before.

Run a Microsoft PowerPoint Presentation

This example demonstrates how to open and run a Microsoft PowerPoint presentation using Automation. This example assumes that you have created a Microsoft PowerPoint presentation named "C:\My Documents\pptPresentation.ppt".

```
Sub runPowerPointPresentation()

    Dim ppApp As PowerPoint.Presentation

    Set ppApp = GetObject("C:\My Documents\pptPresentation.ppt")
    pr.SlideShowSettings.Run

End Sub
```

Creating a PowerPoint Slide Containing a Graphic

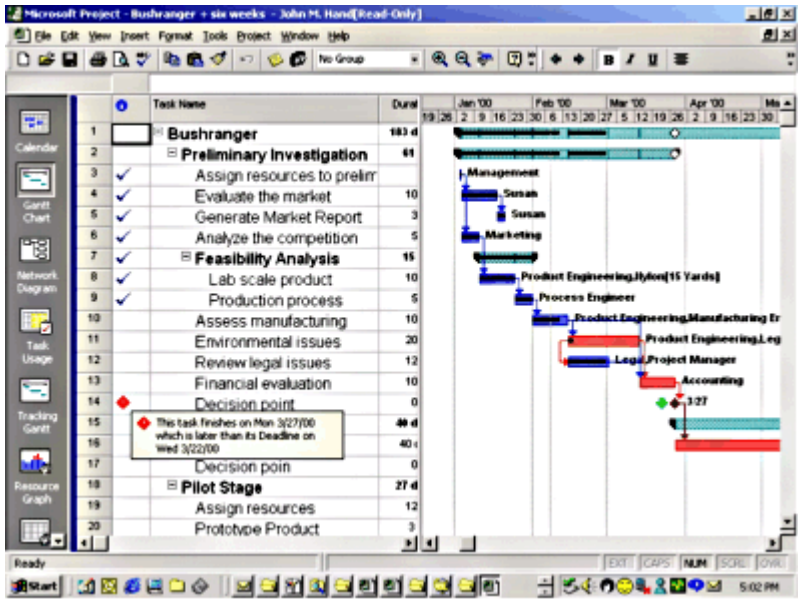
This example demonstrates how to use Automation to create a new PowerPoint slide containing a graphic image.

```
Sub CreateGraphicOnSlide()

   Dim ppApp As PowerPoint.Application
   Dim ppPres As PowerPoint.Presentation
   Dim ppShape As PowerPoint.Shape
   Dim ppCurrentSlide As PowerPoint.Slide

   Set ppApp = CreateObject("PowerPoint.Application")
   ppApp.Visible = True

   Set ppPres = ppApp.Presentations.Add(msoTrue)
   Set ppCurrentSlide = ppPres.Slides.Add(Index:=1, Layout:=ppLayoutText)

   With ppCurrentSlide
      'Set the text of the text frames on the slide
      .Shapes(1).TextFrame.TextRange.Text = "PowerPoint Programmability"
      .Shapes(2).TextFrame.TextRange.Text = "Sixteen Point Star"

      'Bring the text frames to the front, so the
      'graphic doesn't hide them
      .Shapes(1).ZOrder msoBringToFront
      .Shapes(2).ZOrder msoBringToFront

      'Add the 16 point star graphic shape
       Set ppShape = .Shapes.AddShape( _
          Type:=msoShape16pointStar, _
          Left:=50, _
          Top:=50, _
          Width:=500, _
          Height:=500)

      .Shapes(3).Fill.PresetTextured msoTextureWovenMat

      'Send the graphic to back so we can see the
      'text frames
      .Shapes(3).ZOrder msoSendToBack
   End With

   'Save the presentation and exit Microsoft PowerPoint
   ppPres.SaveAs "c:\My Documents\pptExample2", ppSaveAsPresentation
   ppApp.Quit
   Set ppApp = Nothing

End Sub
```

Microsoft Project

Build a Project Plan

Microsoft Project 2000 is a powerful, flexible tool designed to help you manage a full range of projects. Schedule and closely track all tasks—and use Microsoft Project Central, the Web-based companion to Microsoft Project 2000, to exchange project information with your team and senior management.



Microsoft Project 2000 helps you get started by creating a working schedule with information you provide on tasks, resources, and costs.

Powerful new features include custom outline codes that allow you to create a project outline structure tailored to your company's work breakdown structure.

Build a Project File from a Database

This example requires a Microsoft Access database with one table containing four fields:

- intTableID
- strTaskName
- dtmStartDate
- strDuration

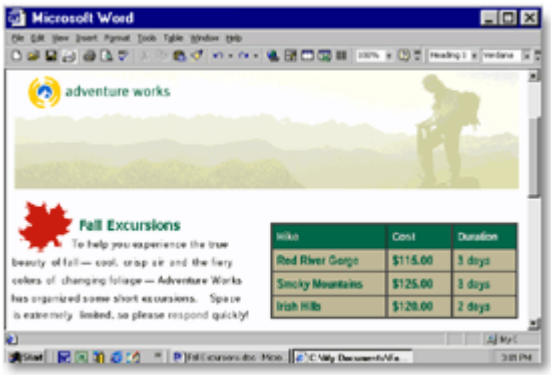
```
Sub ProjectTest()
Dim db As DAO.Database
Dim rst As DAO.Recordset
Dim fMoreThanOne As Boolean
Dim tempTask As String
Dim tempStartDate As String
Dim tempFinishDate As String
Dim objProject As MSProject.Application

' use this constant to hold the pathname for the Project file
Const PROJECTFILE = "C:\My Documents\TestProject.mpp"

    ' remove the previous example file
    If Dir(PROJECTFILE) <> "" Then
        Kill PROJECTFILE
    End If

    Set db = CurrentDb
    Set rst = db.OpenRecordset("TestTable", dbOpenSnapshot)

    Set objProject = CreateObject("MsProject.Application")
    objProject.FileNew False

    ' as this is a brand new file we know that there are no
    ' existing tasks
    fMoreThanOne = False

    Do Until rst.EOF

        With rst
            tempTask = !strTaskName
            tempStartDate = CStr(!dtmStartDate)
            tempDuration = !strDuration
        End With

        With objProject

            ' SelectTaskCell uses relative positioning when the
            ' Row argument is used; don't use Row for the first task
            If fMoreThanOne = True Then
                .SelectTaskCell Row:=1
            Else
                .SelectTaskCell
                fMoreThanOne = True
            End If

            ' enter the name for the summary line
            .SetTaskField Field:="Name", Value:=tempTask

            .SetTaskField Field:="Start", Value:=tempStartDate
            .SetTaskField Field:="Duration", Value:=tempDuration
```

```
        End With
        rst.MoveNext
    Loop
    rst.Close
    db.Close
    Set db = Nothing

    ' save the baseline before attempting to save the file
    objProject.BaselineSave All:=True
    objProject.FileSaveAs Name:=PROJECTFILE, FormatID:="MSProject.MPP.8"
    objProject.Quit
    Set objProject = Nothing

End Sub
```

Microsoft Word

The Microsoft Office Word Processor

Microsoft Word 2000 gives you the tools to more easily create professional-quality documents and share information—in print, e-mail, and on the Web.



The World's Best-Selling Word Processor for Microsoft Windows® Is Now Better Than Ever

Word 2000 combines streamlined document creation with powerful Web functionality so you can work more efficiently and communicate your ideas more effectively. Advanced integration with the rest of Office 2000 enables you to easily include text, data, and graphics from other Office applications to create high-impact documents.

Automate Word with the New Keyword

This is the basic code to create an instance of Word using the New keyword.

```
Sub automateWord()

   Dim wordApp As Word.Application

   ' Create new hidden instance of Word.
   Set wordApp = New Word.Application
   ' Show this instance of Word.
   wordApp.Visible = True

    With wordApp
        ' Code to automate Word here.
    End With

    wordApp.Quit
    Set wordApp = Nothing
End Sub
```

Create a Word Document

The following code creates a document and records its creation date.

```
Sub createDoc()

   Dim wordApp As Word.Application
   Dim wordDoc As Word.Document
   Dim wordRng As Word.Range
   Dim wordPara As Word.Paragraph

   Set wordApp = CreateObject("Word.Application")

   With WordApp

      .WindowState = wdWindowStateMaximize
      .Documents.Add
      Set wordDoc = wordApp.ActiveDocument
      Set wordRng = wordDoc.Range

      With wordRng

         .Font.Bold = True
         .Font.Italic = True
         .Font.Size = 16
         .InsertAfter "Running Word Using Automation"
         .InsertParagraphAfter

         ' insert a blank paragraph between the two paragraphs
         .InsertParagraphAfter

      End With

      Set wordPara = wordRng.Paragraphs(3)

      With wordPara.Range

         .Bold = True
         .Italic = False
         .Font.Size = 12
         .InsertAfter "Report Created:  "
         .Collapse Direction:=wdCollapseEnd
         .InsertDateTime DateTimeFormat:="MM-DD-YY HH:MM:SS"

      End With

      .ActiveDocument.SaveAs "c:\My Documents\createDoc.Doc"
      .Quit

   End With

   Set wordPara = Nothing
   Set wordRng = Nothing
   Set wordDoc = Nothing
   Set wordApp = Nothing

End Sub
```

Closing the Microsoft Word Document

Through Automation, it is possible to close the files you are working with using the Close method. In order to destroy the Automation object variable and close the instance of the application, use the Quit method and set the object variable to the keyword Nothing.

When the Automation object variable goes out of scope, the instance of Microsoft Word is unloaded unless the object was created from a previous instance. It is possible to set the object to a static or public variable so it does not lose scope until the application is closed.

```
Sub CloseWordDoc()

    Dim WordApp As Word.Application
    Dim WordDoc As Word.Document

    'Open an instance of Word
    Set WordApp = CreateObject("Word.Application")

    With WordApp
        Set WordDoc = .Documents.Open("C:\My Documents\Test.Doc")
        'Selects the entire document and makes it Bold

        With WordDoc
            .Range.Font.Bold = True
            'Closes the Document and saves changes
            .Close (wdSaveChanges)
        End With

        .Quit
    End With

    Set WordDoc = Nothing
    Set WordApp = Nothing

End Sub
```

Access a Word Document

This function opens the document created with the Create a Document example, counts the words and returns the word count and document name.

```
Function accessWordDoc(docName)

   Dim wordApp As Word.Application
   Dim wordDoc As Word.Document

   ' Create new hidden instance of Word.
   Set wordApp = New Word.Application

   Set wordDoc = wordApp.Documents.Open(FileName:=docName)

   ' Display document name and count of words, and then close
   ' document without saving changes.
   With wordDoc
       accessWordDoc = "'" & .Name & "' contains " & .Words.Count & " words."
       .Close wdDoNotSaveChanges
   End With

   wordApp.Quit
   Set wordApp = Nothing

End Function
```

Run this example by entering the following line in the Immediate window:

```
   ?accessWordDoc("c:\My Documents\createDoc.Doc")
```

and press Enter.

Inserting Data into a Microsoft Word Document

With Automation code, you can open a Microsoft Word 2000 document and move to a bookmark location in the document. The following example opens a Microsoft Word document and inserts text after a bookmark.

This example assumes that you have Microsoft Word 2000 on your computer, that you have an existing document called C:\My Documents\WordTest.doc, and that the document contains a pre-defined bookmark named City.

```
Sub FindBMark()

   Dim wordApp As Word.Application
   Dim wordDoc As Word.Document
   Dim wordRange As Word.Range

   Set wordApp = CreateObject("Word.Application")
   Set wordDoc = wordApp.Documents.Open("C:\My Documents\Wordtest.doc")

   wordApp.Visible = True

   ' go to the bookmark named "City."
   Set wordRange = wordDoc.Goto(What:=wdGoToBookmark, Name:="City")
   wordRange.InsertAfter "Los Angeles"

   ' print the document.
   wordDoc.PrintOut Background:=False

   ' save the modified document.
   wordDoc.Save

   ' quit Word without saving changes to the document.
   wordApp.Quit SaveChanges:=wdDoNotSaveChanges

   Set wordApp = Nothing

End Sub
```

Find a Bookmark in an Embedded Microsoft Word Document

By using Automation code in Microsoft Access, you can open a Microsoft Word 2000 document and move to a bookmark location in the document.

The following example opens a document that is embedded in a Microsoft Access form.

This example assumes that you have Microsoft Word 2000 set up on your computer, that you have a document called C:\My Documents\WordTest.doc, and that the document contains a pre-defined bookmark called City.

1. Open the sample database Northwind.mdb.
2. Open any module in Design view.
3. On the Tools menu, click References.
4. Click Microsoft Word 2000 Object Library in the Available References box. If that selection does not appear, click the Browse button and look for a file called Msword9.olb, which is installed in the C:\Program Files\Microsoft Office\Office folder by default.
5. Click OK in the References dialog box.
6. Create a new form not based on any table or query in Design view.
7. Add an unbound object frame control to the detail section of the form.
8. When the Insert Object dialog box appears, click Create From File, and then click the Browse button to select your C:\My Documents\WordTest.doc file.
9. Click Open in the Browse dialog box, and then click OK in the Insert Object dialog box.
10. Set the following properties for the unbound object frame control:
11.
12. ```
    Unbound Object Frame
    Name: UnboundObj
    Locked: No
    ```
15. Add a command button to the form; set its Name property to EditWordDoc and set its OnClick property to the following event procedure:
16.
17. ```
    Private Sub EditWordDoc_Click()
    ```
18.
19. ```
        Dim wordApp As Word.Application
        Dim wordDoc As Word.Document
        Dim wordRange As Word.Range
    ```
22.
23. ```
        ' Open Microsoft Word 2000 in place and activate it.
        Me![UnboundObj].Verb = -4
        Me![UnboundObj].Action = 7
    ```
26.
27. ```
        Set wordApp = Me![UnboundObj].Object.Application
        Set wordDoc = wordApp.ActiveDocument
        Set wordRange = wordDoc.Goto(What:=wdGoToBookmark, Name:="City")
        wordRange.InsertAfter "Los Angeles"
        wordApp.Quit
        Set wordApp = Nothing
    ```
33.
34. ```
    End Sub
    ```
35. Save the form as frmBookmark, and then open it in Form view.
36. Click the command button on the form and note that the document is edited in place on the form, and that the words Los Angeles are inserted after the City bookmark.

Sending the Current Record to Word.

The following example uses bookmarks in a Microsoft Word document to mark the locations where you want to place data from a record on a Microsoft Access form.

Creating a Microsoft Word Document

1. Start Microsoft Word and create the following new document:

   First Last
   Address
   City, Region, PostalCode

   Dear Greeting,

   Northwind Traders would like to thank you for your employment during the past year. Below you will find your photo. If this is not your most current picture, please let us know.

   Photo

   Sincerely,
   Northwind Traders

2. Create a bookmark in Microsoft Word for the words "First," "Last," "Address," "City," "Region," "PostalCode," "Greeting," and "Photo":
   a. Select the word "First."
   b. On the Insert menu, click Bookmark
   c. In the Bookmark Name box, type "First," (without the quotation marks) and then click Add.
   d. Repeat steps 2a through 2c for each of the remaining words, substituting that word for the word "First" in steps 2a and 2c.

3. Save the document as C:\My Documents\MyMerge.doc, and then quit Microsoft Word.

Sending Data to Microsoft Word from a Microsoft Access Form

1. Start Microsoft Access and open the sample database Northwind.mdb.
2. Set a reference to the Microsoft Word 9.0 Object Library. To do so, follow these steps:
   a. Open any module in Design view.
   b. On the Tools menu, click References.
   c. Click Microsoft Word 9.0 Object Library in the Available References box. If that selection does not appear, browse for Msword9.olb, which installs by default in the C:\Program Files\Microsoft Office\Office folder.
   d. Click OK.
   e. Close the module.

3. Open the Employees form in Design view.
4. Add a command button to the form and set the following properties:
5.
6. `Command Button:`
7. `Name: MergeButton`
8. `Caption: Send to Word`
9. `OnClick: [Event Procedure]`
10. Set the OnClick property of the command button to the following event procedure.
11.

```
12. Private Sub MergeButton_Click()
13. 
14.     On Error GoTo MergeButton_Err
15.     Dim wordApp As Word.Application
16. 
17.     ' copy the Photo control on the Employees form.
18.     DoCmd.GoToControl "Photo"
19.     DoCmd.RunCommand acCmdCopy
20. 
21.     ' start Microsoft Word.
```

```
    Set wordApp = CreateObject("Word.Application")

    With wordApp

       ' Make the application visible.
       .Visible = True
       ' Open the document.
       .Documents.Open ("c:\My Documents\myMerge.doc")
       ' Move to each bookmark and insert text from the form.
       .ActiveDocument.Bookmarks("First").Select
       .Selection.Text = (CStr(Forms!Employees!FirstName))
       .ActiveDocument.Bookmarks("Last").Select
       .Selection.Text = (CStr(Forms!Employees!LastName))
       .ActiveDocument.Bookmarks("Address").Select
       .Selection.Text = (CStr(Forms!Employees!Address))
       .ActiveDocument.Bookmarks("City").Select
       .Selection.Text = (CStr(Forms!Employees!City))
       .ActiveDocument.Bookmarks("Region").Select
       .Selection.Text = (CStr(Forms!Employees!Region))
       .ActiveDocument.Bookmarks("PostalCode").Select
       .Selection.Text = (CStr(Forms!Employees!PostalCode))
       .ActiveDocument.Bookmarks("Greeting").Select
       .Selection.Text = (CStr(Forms!Employees!FirstName))
       ' Paste the photo.
       .ActiveDocument.Bookmarks("Photo").Select
       .Selection.Paste

    End With

    ' print the document in the foreground so Word
    ' will not close until the document finishes printing.
    objWord.ActiveDocument.PrintOut Background:=False

    ' Close the document without saving changes.
    objWord.ActiveDocument.Close SaveChanges:=wdDoNotSaveChanges

    ' Quit Microsoft Word and release the object variable.
    wordApp.Quit
    Set wordApp = Nothing
    Exit Sub

MergeButton_Err:
    ' If a field on the form is empty
    ' remove the bookmark text and continue.
    If Err.Number = 94 Then
       objWord.Selection.Text = ""
       Resume Next
       ' If the Photo field is empty.
    ElseIf Err.Number = 2046 Then
       MsgBox "Please add a photo to this record and try again."
    Else
       MsgBox Err.Number & vbCr & Err.Description
    End If
    Exit Sub
End Sub
```

77. Save the Employees form and open it in Form view.
78. Click the Send To Word button to start Microsoft Word, merge data from the current record on the form into MyMerge.doc, print the document, and then close Microsoft Word.

NOTE: When you use this method of inserting text into a Word Document, you are deleting the bookmark

when you insert the record field content. If you need to reference the text that you entered into the document, you must bookmark it. You can use the following sample to add the bookmark "Last" to the text inserted from record field "LastName."

```
.ActiveDocument.Bookmarks("Last").Select
.Selection.Text = (CStr(Forms!Employees!LastName))

' add this line to reapply the bookmark name to the selection
.ActiveDocument.Bookmarks.Add Name:="Last",Range:=Selection.Range
```