Much ADO about JET

Tim Anderson on working with JET and the Advanced Data Objects API.

Data access is fundamental to most applications and for some time Microsoft has been pushing OLEDB, its universal data access strategy, as the way to do it. At a higher level, this means using an API called ADO which used to stand for ActiveX Data Objects but now means Advanced Data Objects, causing developers to wonder again whether Microsoft has a dedicated Name Confusion Department. Whatever, there have been problems with ADO.

One of the frustrations has been the scarcity of native OLEDB drivers, forcing the use of ODBC underneath the OLEDB wrapper. This has particularly dire consequences when using JET, the database engine of Access and Visual Basic, since ODBC to JET gives notoriously poor performance.

JET now has a native OLEDB driver, much improved in JET 4.0, and ADO is now at version 2.1. If you need programmatic access to data, this is definitely the way to go. Incidentally, JET 4.0 is stuffed with new features and represents a significant advance on earlier versions, as I discovered when hearing Microsoft's Kevin Collins speak on the subject at the Microsoft Office and VBA Solutions (MOVS) conference in London.

For instance, JET now handles row-level locking as an option and the maximum database size has doubled, to 2Gb.

Some of the tips that follow come from this conference and you can find out more at www.vbaconference.com.

Getting up-to-date

The safest way to install the latest database features is to go to www.Microsoft.com/data and download the MDAC (Microsoft Data Access) SDK which, at the time of writing, is version 2.1. Alternatively, you may find that installing Internet Explorer 5.0 gets you all you need, and certainly Office 2000 will get you up and running.

JET is used extensively by numerous Microsoft products including, oddly enough, SQL Server so you will not find many Windows systems that do not have it installed.

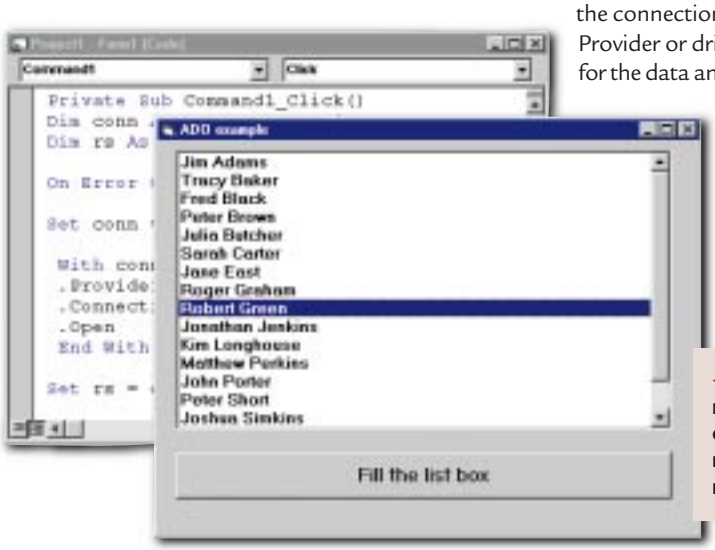
What ADO looks like

Work with ADO generally starts with a Connection object. The property ConnectionString sets up the connection by defining the Provider or driver, the file name for the data and, possibly, other parameters. Then you can call on the Connection's Open method to make the connection live.

[FIG 1] Fill a listbox

```
Using ADO in code
Private Sub Command1_Click()
Dim conn As ADODB.Connection
Dim rs As ADODB.Recordset

On Error GoTo ErrHandler
Set conn = New ADODB.Connection
 With conn
 .Provider = "Microsoft.Jet.OLEDB.4.0"
 .ConnectionString = "data source =↙
 C:\MYDATA\SPORTS.MDB"
 .Open
 End With
Set rs = conn.Execute("Select * from↙
 members order by lastname")
rs.MoveFirst
Do While Not rs.EOF
List1.AddItem (rs!Firstname + " " +↙
 rs!LastName)
List1.ItemData(List1.NewIndex) = rs!ID
rs.MoveNext
Loop
rs.Close
conn.Close
Exit Sub

ErrHandler:

MsgBox Err.Description

End Sub
```

(Key: ↙ Code string continues)



◀FIG 2 YOU DON'T NEED BOUND CONTROLS TO ACCESS DATA. IT CAN ALL BE DONE IN CODE

Once you have a connection, you would typically create a Command object and set its ActiveConnection property to the live connection. Next, you can set the CommandText property to an SQL query and then call Execute to return a Recordset. Another option is simply to call the Execute method of a Connection object to get hold of a Recordset without the need for a separate Command object. These Recordset objects will look familiar if you have worked with the old-style Data Access Objects. A Recordset enables you to read and write data.

Fig 1 puts this together in a simple example; filling a listbox with the contents of a database table. This

uses the native JET OLEDB provider. Note that for this to work you need first to open up the Project References dialogue and select your latest version of the Microsoft ActiveX Data Objects library — despite the name change, the dialog still says ActiveX and not 'Advanced'.

You might wonder why you should write this code to fill a list box when you can achieve the same result with a Datacontrol and a DBList?

It is partly a matter of style. Bound controls have advantages and are quicker to set up, but manual coding gives the developer complete control over the data access [Fig 2].

Many developers use both approaches. In a real-world database application you will need to manipulate the data purely in code at some point, so it pays to understand how to do it. This is especially true if you are coding for the web, or for any multi-tier application where the data access code is detached from the user-interface.

ADO tips and tricks

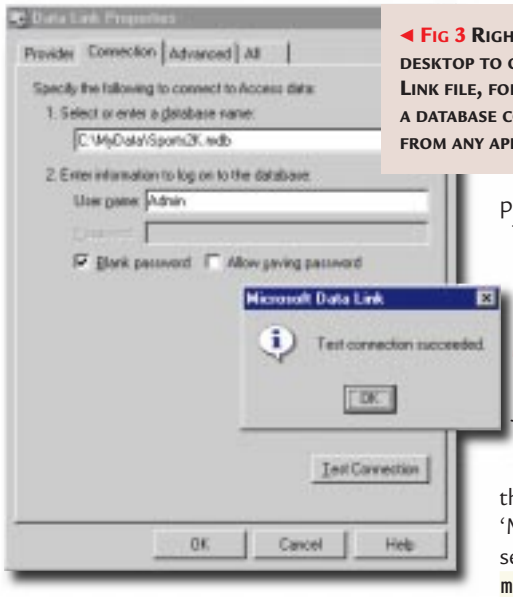
Universal Data Links.

Making a data connection in Windows is only a right-click away. A Universal Data Link file is a small text file that defines a connection. If you right-click the desktop, or drop-down the File menu in Explorer, and choose New, Microsoft Data Link, a new UDL file is created. Then, you can double-click the new file to define the connection [Fig 3].

By default it will use the ODBC provider but you can easily select a native OLEDB provider instead. Define and test the connection using the tabbed dialogue and rename the file as required — for instance, to MyConn.UDL. Now you can use the connection in your applications. For example:

```
Dim MyConnection as ADODB↙
.Connection
Set MyConnection = New ADODB↙
.Connection
MyConnection.Open "File Name↙
=C:\MyConn.UDL;"
...
```

(Key: ↙ Code string continues)



◄ **Fig 3 Right-click the desktop to create a Data Link file, for configuring a database connection from any application**

Note that ADO does not like to see spaces between the File Name parameter and the name of the file.

The advantage of a UDL file is that you can re-use this same connection in any application that supports ADO.

Connect directly to Outlook and Excel.

JET 4.0 comes with drivers for Outlook and for Excel. To try this, create a new connection and set the Provider to 'Microsoft.Jet.OLEDB.4.0'. Then set the ConnectionString to, say:

```
myconn.ConnectionString =↙
"Data Source = C:\MySheet↙
.xls; Extended Properties =↙
Excel 8.0"
```

(Key: ↙ Code string continues)

Open the connection and you can traverse the rows of the spreadsheet as if they were records in a database table. Now, if your database is actually an Excel spreadsheet it is not altogether obvious what are the field names and table names. The way to find out is to use a schema rowset.

Use schemas to discover information.

A schema rowset is a means of querying the database for information about itself. To take the above example, here is how you could discover the table names in an Excel workbook, or indeed any database:

```
Dim rs as ADODB.Recordset
Set rs = myconn.OpenSchema↙
(adSchemaTables)
 Do Until rs.EOF
 ListBox1.AddItem "" &↙
 rs!TABLE_NAME
 rs.MoveNext
 Loop
rs.Close
```

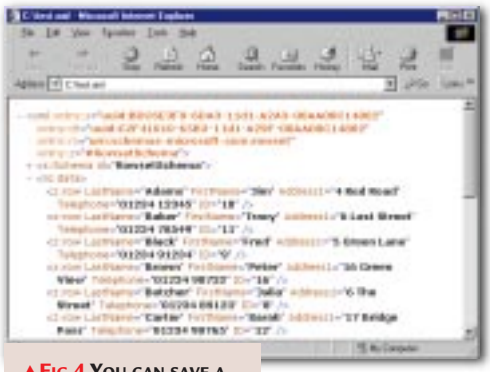
(Key: ↙ Code string continues)

The exact range of schemas available varies according to the data provider you are using but it is generally extensive. One handy schema rowset in JET 4.0 enables you to get the list of current users, with `DBSCHEMA_JETOLEDB_USERROSTER`

To obtain this you need to pass a GUID to the OpenSchema method, as detailed in the documentation for the JET provider.

Save a recordset to disk and load it back later.

The ADO Recordset object has a Save method. This does not, as you might expect, update the database with changed values; this is the role of the Update method. Instead, the Save method saves the whole recordset as a file. For this to work, you must set the Connection's CursorLocation property to adUseClient, which means that the cursor, or set of records, is managed on the client workstation and not on the database server.



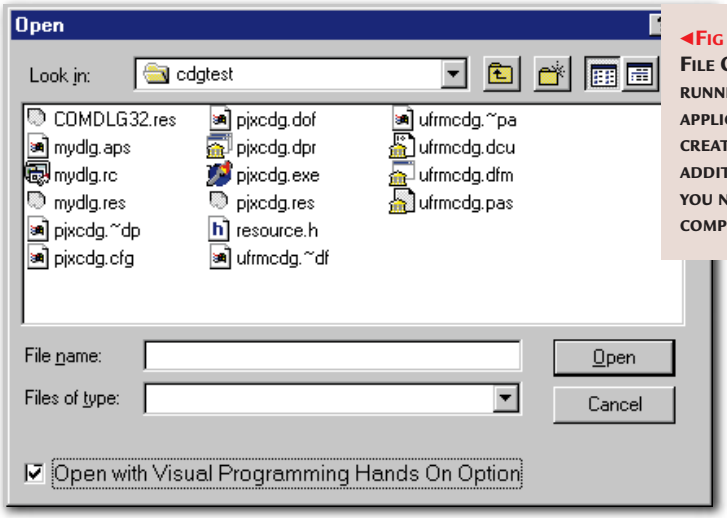
▲ **Fig 4 You can save a Recordset as XML and view it in Internet Explorer 5.0**

This technique opens up numerous possibilities. For example, you could connect your laptop to a network and run a small piece of code to query a server database, and save the results to disk. You could then take the laptop on the road and have the application load the recordset from disk, giving you access to the data without actually making a connection. You can do it using a few lines of code:

```
Dim rs as ADODB.Recordset
Set rs = New ADODB.Recordset
rs.Open("C:\MyRs.adtg")
...
```

Note that in this case you do not need a Connection object; it just works.

Another neat trick [Fig 4] is to save the recordset as XML :



◀FIG 5 THE MODIFIED FILE OPEN DIALOGUE RUNNING IN A DELPHI APPLICATION. TO CREATE THE ADDITIONAL CONTROL, YOU NEED A RESOURCE COMPILER

```
rs.Save("C:\MyRs.XML",↙
adPersistXML)
```

(Key: ↙ Code string continues)

In conjunction with an XML stylesheet, this would let you open the recordset, nicely formatted in a browser. As the XML standard gathers momentum, it will also let you easily exchange data with a variety of databases running on different platforms.

Use Borland Database Engine. This is called returning the favour. For some time now, the Borland Database Engine, which is used by Delphi, dBase and Paradox has allowed you to open JET databases, which it does by using DAO. Now, you can also open dBase and Paradox tables from JET. Previously this had been possible using Microsoft's drivers for these formats but the capabilities of the drivers has fallen way behind the latest versions of dBase and Paradox. The new driver simply calls the Borland Database Engine if it is available, so that the latest formats are supported.

Tinkering with common dialogues

Reader Kevin Parsons asks how you can modify the standard Windows dialogues using Delphi. He is trying to recreate the effect seen in the Open dialogue of Paradox 7, where a combo-listbox is added to the mix of components and is used to list all of the BDE aliases.

This task falls into two parts. Firstly, you have to add the required controls to the common dialogue box. Secondly, you need to control them with Windows messages.

The common dialogues use dialogue resources, so to modify them you need a resource editor like the one that comes with Visual C++. You can either replace the dialogue resource template using the original as a starting point, or you can create a template with just the controls you want to add, setting the WS_CHILD and WS_CLIPSIBLINGS style. Save this as a .RES resource file.

Next, you need to create your own common dialogue class as a descendant of TOpenDialog. In the constructor, set the Template property to your custom template with, for example;

```
self.template = MakeIntResource(n);
```

where n is the resource identifier for your dialogue.

Next, you need to add custom processing to the messages sent by the dialogue controls, including initialising them in response to WM_INITDIALOG.

Whenever a selection changes, a WM_NOTIFY message is sent. Looking at Delphi's documentation, it appears that the MessageHook function is the one you need to override, although if you study the source in Dialogs.pas it appears that this is not, in fact, the hook procedure for the dialogue.

Instead, dialogue messages are processed in the WndProc procedure and you will need to override this to intercept the required messages. I used this technique to add a custom checkbox to the file open dialogue [Fig 5].

To make sense of all this, you will need to refer to the Windows Platform SDK as well as the Delphi source code. Another option is to build your own File Open dialogue using Delphi components.

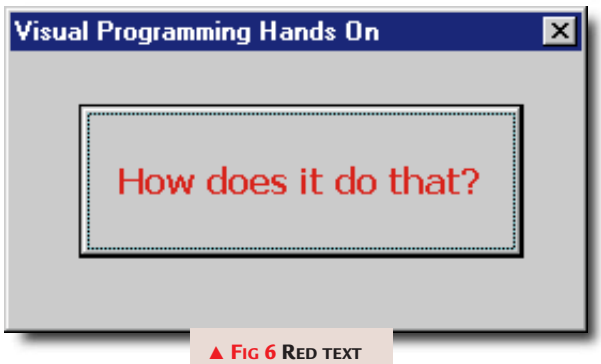
Colour that button!

Reader Trevor Hackworth complains that in VBA, when he puts command buttons on a form, although he can easily change the backcolor and forecolor properties, in VB5 and VB6 the forecolor is not available. He would like to set the text colour on individual command buttons to different colours on the same form.

This is one of the hidden, dark recesses of Windows. The clue is there in the SDK under WM_CTLCOLORBTN: '*The text colour of a push button applies only to its focus rectangle; it does not affect the colour of the text.*'

Visual Basic protects you from this frustration by not allowing you to set the Forecolor property. Incidentally, in Delphi you can set the colour of the font but it has no effect. The SDK goes on to suggest that you can use an owner-drawn button if you want a special appearance — it is not an easy option in Visual Basic [Fig 6].

So why does it work in VBA? One possibility is that VBA is drawing the caption on the button surface using the API text functions.



▲ FIG 6 RED TEXT ON A BUTTON, BUT ONLY IN VISUAL BASIC FOR APPLICATIONS

You can do the same in VB, of course, but unfortunately the surface gets redrawn and blanked out whenever it is clicked. Like most things in VB, there will be a way around it with some API trickery, but it is not that easy. One idea would be to create images with the button captions on and use a graphical button style with a picture.

PCW CONTACTS

Tim Anderson welcomes your Visual Programming comments and queries. Contact him at visual@pcw.co.uk or via the PCW editorial office (address p10).