Visual Basic 4.0 Technical Articles

Introduction to Using the Remote Data Object

Ken Lassesen
Microsoft Developer Network Technology Group

September 11, 1995

Abstract

This technical article shows an experienced Visual Basic® 3.0 developer how to exploit the Remote Data Object (RDO) available in Visual Basic 4.0 Enterprise Edition. This article allows the developer to move existing applications from using the DAO to using the RDO quickly.

> Note The RDO is only available in 32-bit Visual Basic. The client must run Windows® 95 or Windows NT® version 3.51 or later.

What Are the RDO and the RDC?

The Remote Data Object (RDO) and Remote Data Control (RDC) are examples of yet other data access methods. The RDC binds the RDO to a control in much the same way that the Remote Data Control binds the Data Access Object (DAO) to a control. The RDO is a thin layer of code placed on top of Open Database Connectivity (ODBC) with some special features like server-side cursors for accessing Microsoft® SQL Server version 6.0. When I first heard about the RDO, I remarked, "O lovely data! How do I access thee? Let me count the ways—DAO, data control, DB-Library™, Jet, VBSQL, ODBC, SQL-DMO . . . ." The number of data access methods available is awesome to some, irritating for the indecisive, and terrifying to the timid. Because many of you may not know what all of these terms, acronyms, and initialisms are, I have listed them in Table 1.

Table 1. Terms, Acronyms, and Initialisms Used for Data Access Methods

Term/Acronym/ Initialism	Description
DAO	Data Access Object
Data Control	Data Control
DBLib	DB-Library
Jet	Microsoft Jet Database Engine
ODBC	Open Database Connectivity
RDO	Remote Data Object
SQL-DMO	SQL Distributed Management Objects
VBSQL	Visual Basic Library for SQL Server (VBSQL), an implementation of the DB-Library application programming interface (API)
RDC	Remote Data Control

Each data access method has strengths and weaknesses. Although there are many factors to consider, my own opinion of how the common methods relate to each other is shown in Figure 1. Actual performance can vary greatly depending on environmental factors—database type, network, amount of traffic, configuration, so take it all with several grains of salt.

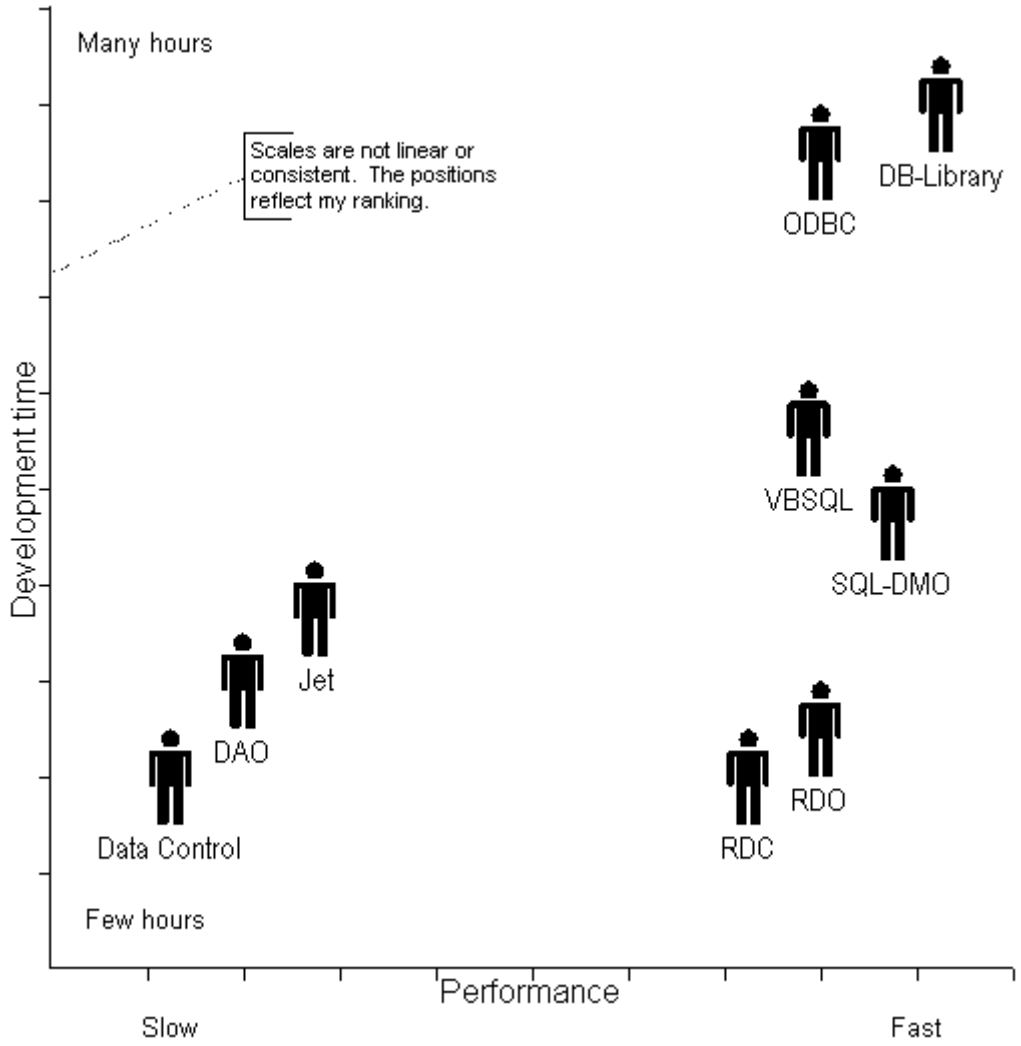

Figure 1. Personal evaluation of common data access methods

To better understand the processes involved, you should examine Figure 2, where I show the layers involved in each of the object-oriented methods.

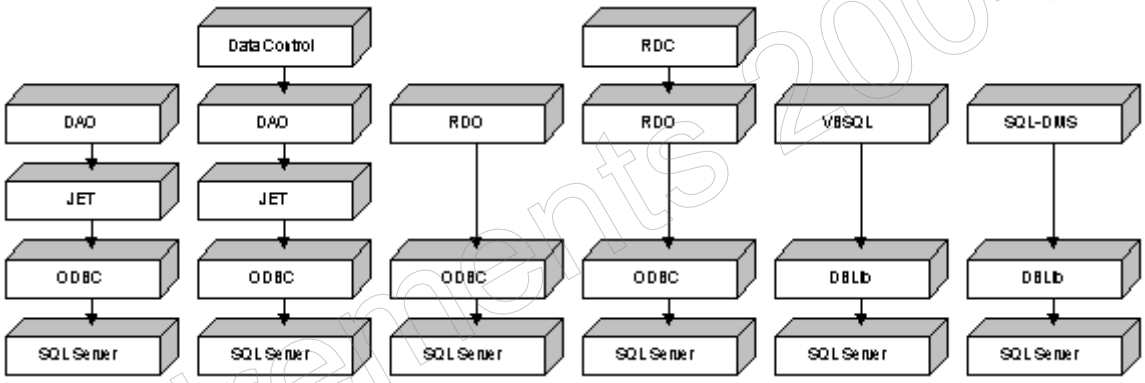

Figure 2. The layers used in common data access methods to communicate with Microsoft SQL Server

The two most critical factors in successful data access are the time it takes to code the application (both development and maintenance coding) and the speed of retrieving data from the database server. In my opinion,

the RDO provides the best mix of those two factors—and the RDO is even better because it requires very few resources from the workstation.

Why Should I Move to Using the RDO from the DAO?

The RDO is an alternative that in many scenarios performs better than the DAO. Some future version of the DAO should include all of the positive features of the RDO, but if you want those features today, you must use the RDO. The RDO's developers designed it explicitly for accessing data on remote servers like Microsoft SQL Server version 6.0 or ORACLE®. The DAO's developers designed it for accessing data from both remote servers and local workstations, for example, from xBase files, Microsoft Access files, and Paradox™ files.

The RDO is a bad choice for reading data from a Microsoft Access database on your local drive because it will go through all of the layers that the DAO uses, as well as its own layer. The RDO is an excellent choice for reading data from Microsoft SQL Server 6.0 because the RDO developers put extra effort into tuning its performance with this database. Think of the RDO as a replacement for making DB-Library™ application programming interface (API) calls or ODBC API calls—it's a way to get performance without spending weeks coding API calls. I repeat, if you are writing client/server applications with Microsoft SQL Server, the RDO is my first choice and I am willing to debate anyone who suggests a different approach.

The greatest advantage of the RDO is that it does not use the Jet engine. The Jet engine consumes a large amount of resources and adds a lot of overhead. The RDO uses a thin code layer over the ODBC layer and the driver manager that establishes connections to the database. This thin layer appears very similar to the DAO but without all of the resources required by Jet on the client PC. This small memory footprint results in a huge drop in the amount of memory required and reduces the code swapped to the hard drive on some workstations. The reduction of disk swapping can produce very significant performance gains on low-end systems, such as a 386SX-25 with 4 MB or less of RAM.

The biggest drawback of the RDO occurs when the developer must convert an application from the DAO code to the RDO code. With a later version of Visual Basic, the developer may have to convert the application back to using the DAO. This drawback is minor—you can use a Visual Basic add-in to do most of the work. I describe add-ins in my article "Building Add-Ins for Visual Basic 4.0."

What does the developer gain or lose when she or he moves from using the RDO to using the DAO? Table 2 shows my opinion of the most significant strengths and weaknesses of each.

Table 2. Comparison of DAO and RDO Features

Feature	DAO	RDO
Resources on workstation	High (Minimal PC: 486DX-66, 12 MB)	Low (Minimal PC: 386DX-33, 6 MB)
Access to server-side cursors	No	Yes
Code is forward-compatible	Yes	No
Asynchronous queries	No	Yes
Multiple result set processing	No (See Note)	Yes
Stored procedure support with output parameters and return values	No	Yes

Note The DAO can technically do multiple result sets; however, the code is ugly. See Bill Vaughn's book, A Hitchhiker's Guide to Visual Basic 4 and SQL Server 6, for more information.

Using the RDO Instead of the DAO

In the following sections, I will take an example of a typical Visual Basic 3.0 routine written using the DAO and then show the same routine written using the RDO. Most Visual Basic developers are very familiar with the DAO and will try to code the RDO as if it were the DAO. Unfortunately, there are some significant coding differences between these two products that can become confusing. The examples should guide you through these difficulties.

The RDO talks to the database at a lower level than the DAO, so you will need to know more about concepts like

connections to take full advantage of the RDO. The lower level gives you more control but with the usual cost of having more to consider. One sweet thing about using connections is that you can set different property values for each one. For example, on a former project, I was using the DAO to work concurrently with Microsoft SQL Servers located in Ireland and in the next office. The DAO forced me to use one set of time-out values for both servers, but with the RDO I can have different time-out values for each server.

A Simple Procedure

To illustrate the changes, I will use a generic procedure that retrieves data from a database and displays the data in a control. I will start with a Visual Basic 3.0 version of this procedure, which I will convert to a Visual Basic 4.0 DAO procedure to illustrate style changes between versions of Visual Basic. By convert I do not mean that Visual Basic 4.0 requires you to port the procedure for it to work; it will work fine as it is. I mean that I will change the procedure to what I think the Visual Basic 4.0 procedure should look like, which is different from a Visual Basic 3.0 procedure—in other words, to a great procedure instead of one that simply works.

Next, I will convert the procedure from a DAO procedure to an RDO procedure. The RDO does not offer as many options as the DAO, but when applicable, the RDO often performs significantly better.

Visual Basic 3.0 DAO Procedure

This sample procedure is a function that fills any list-box type of control from an SQL query statement. The procedure returns the number of rows added to the control. The sample below shows a typical Visual Basic 3.0 style of coding. (Note that some extra returns have been added—for example, in the first Function statement—to make the code readable on your screen.)

```
Function VB3_FillLstFromSQL(Lst As Control, ByVal DBName$, ByVal SQLQuery$,
  ByVal DBConnect$, ByVal ReadOnly%, ByVal RSType%, ByVal RSOptions%)
On Error GoTo VB3_Error
Dim DB As Database
Dim RS As Recordset
Dim nRecords&
VB3Consistent_DAOOpenRecordset DBName$, RSType%, RSOptions, DBConnect$
If ReadOnly% <> 0 Then ReadOnly% = True 'Validate

Lst.Clear
Set DB = OpenDatabase(DBName$, False, ReadOnly%, DBConnect$)
Set RS = DB.OpenRecordset(SQLQuery$, RSType%, RSOptions%)
While Not RS.EOF
    Lst.AddItem (RS(0))
    RS.MoveNext
    nRecords& = nRecords& + 1
Wend

VB3_Exit:
VB3_FillLstFromSQL = nRecords& 'Return # added to Lst
On Error GoTo 0
Exit Function

VB3_Error: 'Display message and exit procedure
MsgBox Error$, vbWarning, "Error:" & Err
Resume VB3_Exit
End Function
```

So what does the procedure do? First, it corrects any inconsistency in the passed parameters by calling VB3Consistent_DAOOpenRecordset. The details are in the IntroRDO sample code. The procedure opens the record set and adds the records. The (RS(0)) converts the first column, which should be the only column, to a string. If the control displays multiple columns, the SQL statement returns a single field that formats these columns as a single field; for example:

```
Select SSN, Lname, Fname From Employees
```

I can submit this query as follows:

```
Select SSN + CHR$(9)+Lname+CHR$(9)+Fname From Employees
```

This query can significantly speed the time it takes to fill the control and can shorten the length of time during which there is a read-lock on the table. If any errors occur, the procedure displays the error and then exits. I reset the error-handling routine when I exit the procedure.

Visual Basic 4.0 DAO Procedure

Visual Basic 4.0 supports several new features, shown in Table 3, that can dramatically change the ease of debugging, the appearance, and the performance of your code. I use the new features because I find my code to be easier to understand and to extend when I do so. The changes are sufficiently significant for me to get up in the pulpit for a short sermon. So just call me Pastor Ken as I show you my favorite new Visual Basic features in Table 3.

Table 3. Recommended New Features to Use in Visual Basic 4.0

Feature	Description

Support for named arguments	Allows arguments to be more meaningful. Instead of:
	`A$=StrTrans(X$,C$,D$)`
	the code reads:
	`A$=StrTrans(Source:=X$, Find:=C$; ChangeTo:=D$)`

Support for optional arguments	Allows arguments to show only essential information and exceptions while supporting more generic procedures.
Supports line continuation	Allows better formatting and commenting of code.
Support for references	Allows for better identification of objects and extends the product.

The Visual Basic 3.0 procedure above in the "Visual Basic 3.0 DAO Procedure" section takes on a very different look when written in Visual Basic 4.0. The change of appearance is due to the use of named arguments and line continuation. The line continuation characters allow long statements to wrap several lines, which makes for better formatting and readability. A programmer can use named arguments to arrange arguments in a different order—for example, from most significant to least significant. Here is the procedure from the previous section written in Visual Basic 4.0:

```
Function DAO_FillLstFromSQL( _
    LstControl As Control, _
    ByVal DBName$, _
    ByVal SQLQuery$, _
    Optional DBConnect, _
    Optional ReadOnly, _
    Optional RSType, _
    Optional Options, _
    Optional Exclusive _
    )
On Error GoTo DAO_Error
Dim DE As New dao.DBEngine
Dim DB As dao.Database
Dim RS As dao.Recordset
Dim nRecords&
VB4Consistent_DAOOpenRecordset _
    DBConnect:=DBConnect, _
    RSType:=RSType, _
    Options:=Options

If IsMissing(ReadOnly) Then
   ReadOnly = True 'Validate
ElseIf Not IsNumeric(ReadOnly) Then
    ReadOnly = True
ElseIf ReadOnly <> 0 Then
    ReadOnly = True
End If
```

```
LstControl.Clear
'I prefer to see Connect and name together:
Set DB = DE.OpenDatabase( _
    Name:=DBName$, _
    Connect:=DBConnect, _
    Exclusive:=Exclusive, _
    ReadOnly:=dbReadOnly _
    )
Set RS = DB.OpenRecordset( _
    Name:=SQLQuery$, _
    Type:=RSType, _
    Options:=Options _
    )
While Not RS.EOF
    LstControl.AddItem (RS(0))
    RS.MoveNext
    nRecords& = nRecords& + 1
Wend

DAO_Exit:
DAO_FillLstFromSQL = nRecords& 'Return # added to Lst
On Error GoTo 0
Exit Function

DAO_Error: 'Display message and exit procedure
MsgBox Error$, vbExclamation, "Error:" & Err
Resume DAO_Exit
End Function
```

The use of optional arguments with named arguments allows the developer to use one procedure in many different ways. This reduces the number of procedures in an application. For example, the VB4Consistent_DAOOpenRecordset procedure uses all optional arguments. This procedure determines the tests and corrections by the arguments passed. The procedure uses the IsMissing function to test for passed arguments. For details on how I implemented it, examine the code in the IntroRDO sample that accompanies this article.

The developer may extend Visual Basic 4.0 very easily by using references, but the use of references requires some caution. Different references may use the same object name, property name, or method name. This may result in names becoming unexpectedly ambiguous. I prevent this ambiguity by always fully qualifying objects. The DAO reference qualifies the Database and Recordset types. This may be overkill, but it's worth it to prevent other people from creating object models with the same name (for example, "Database"). Other database manufacturers may imitate the DAO model to make it easier for developers to use (and force you to add the qualifiers later). I prefer to solve problems before they arrive and have placed the rule "Always fully qualify all objects" beside the rule "Always use Options Explicit" because the reasons for doing so are very similar.

Calling the DAO_FillLstFromSQL procedure can also appear to be very different in Visual Basic 4.0 than it is in Visual Basic 3.0. Visual Basic 3.0 requires the procedure call to specify all arguments. A misplaced argument can cause one to spend long hours debugging a program—which actually happened to me in preparing this article. Here is an example of an equivalent Visual Basic 3.0 procedure call (we added the return to make the line visible on your screen):

```
VB3_FillLstFromSQL lstListBox, (DBName), txtSQLQuery, txtDBConnect,
dbReadOnly, ThisRSType(), ThisRSOptions()
```

Visual Basic 4.0 requires only the arguments that are not the default values, resulting in less muddled code. The naming of variables and named arguments can also clarify code. The Visual Basic 4.0 procedure call below is easier to understand than the call shown above.

```
DAO_FillLstFromSQL _
        LstControl:=lstListBox, _
        DBName:=txtDBName.Text, _
        DBConnect:=txtDBConnect.Text, _
        SQLQuery:=txtSQLQuery.Text
```

Having changed my code to the new Visual Basic 4.0 style, I will now convert this procedure from using the DAO to using the RDO.

Visual Basic 4.0 RDO Procedure

The RDO code is very similar in overall pattern to that of the DAO, although the designers changed some names to confuse the innocent. First I'll show you the RDO code and then explain what has changed and why.

```
Function RDO_FillLstFromSQL( _
    LstControl As Control, _
    ByVal DBName$, _
    ByVal SQLQuery$, _
    Optional DBConnect, _
    Optional ReadOnly, _
    Optional RSType, _
    Optional Options, _
    Optional LockType, Optional Prompt _
    ) 'NOTE This is the line continuation limit
On Error GoTo rDO_Error
Dim DB As rdo.rdoConnection
Dim Env As rdo.rdoEnvironment
Dim RS As rdo.rdoResultset
Dim nRecords&
LstControl.Clear

Consistent_RDOOpenRecordset _
    DBConnect:=DBConnect, _
    RSType:=RSType, _
    Options:=Options

If IsMissing(ReadOnly) Then
   ReadOnly = True 'Validate
ElseIf Not IsNumeric(ReadOnly) Then
    ReadOnly = True
ElseIf ReadOnly <> 0 Then
    ReadOnly = True
End If


Set DB = rdoEngine.rdoEnvironments(0).OpenConnection( _
    dsName:=DBName$, _
    Prompt:=Prompt, _
    ReadOnly:=ReadOnly, _
    Connect:=DBConnect _
    )

Set RS = DB.OpenResultset( _
    Name:=SQLQuery, _
    Type:=RSType, _
    LockType:=LockType, _
    Option:=Options _
    )

While Not RS.EOF
    LstControl.AddItem (RS(0))
    RS.MoveNext
    nRecords& = nRecords& + 1
Wend

rDO_Exit:
RDO_FillLstFromSQL = nRecords& 'Return # added to Lst
On Error GoTo 0
Exit Function

rDO_Error: 'Display message and exit procedure
MsgBox Error$, vbExclamation, "Error:" & Err
Resume rDO_Exit
```

```
End Function
```

The first thing you will notice about the code above is that the procedure declaration drops one argument and adds two more arguments. All of these procedure arguments are intentionally optional so that I can switch easily from the DAO to the RDO if I do not use them. For simplicity, I omitted the asynchronous code from the above example; see the IntroRDO sample for how to handle asynchronous result sets.

The DAO Exclusive argument grants one user-exclusive control over the database (usually for performance reasons), which does not fit the RDO scenario, in which multiple users can access the database.

The RDO LockType argument determines the type of locking on the server. The common RDO.LockType can be one of these three types: optimistic concurrency based on row values, optimistic concurrency based on row ID, or pessimistic concurrency. Changing the type of lock concurrency can result in significant performance gains on database servers with heavy loads by reducing the number and duration of locks.

The RDO Prompt argument grants more flexibility in how the ODBC connection occurs. Next, I will examine the differences between RDO.rdoConnection and DAO.OpenDatabase, between RDO.rdoResultset and DAO.Recordset, and between RDO.rdoPreparedStatement and DAO.QueryDef.

The DAO and RDO Equivalents

Before getting into the details, I would suggest printing copies of the DAO and RDO maps for reference. You will find these maps in my articles "Mapping the Data Access Object: DAO 3.0" and "Mapping the Remote Data Object: RDO 1.0." Your first impression will be of how small an object RDO is! This smallness is because the RDO developers aimed it for a client/server application. They assumed that the developer would use other tools, such as SQL-DMO, to define tables, triggers, user log-ins, and views. In short, the RDO provides only an application interface and excludes parts of a development interface.

RDO.rdoConnection and DAO.OpenDatabase

The RDO.rdoConnection object provides functionality similar to that of the DAO.Database object. The major difference between these objects is the ability of the DAO to directly open a physical database file; the RDO cannot directly open a physical database file. Both of these functions share the ODBC Connect property. The RDO.rdoConnection requires an environment that is a new layer that the DAO does not have. Usually the default environment is adequate unless you wish to do a single transaction against multiple servers or need to sign on as several different users concurrently.

Another big difference between these two objects is that the RDO.rdoConnection object represents a physical connection to the server. When you call the RDO.Close method on the RDO.rdoConnection object, the RDO calls SQLDisconnect and drops the connection immediately. This is different from the DAO, where Jet will cache connections for reuse that can cause complex problems if one of these cached connections dies unexpectedly.

Speaking of connections, if you do not close the connection when using the RDO, the RDO adds the result set to the RDO.rdoResultsets collection. You can locate a result set in this collection either by ordinal number or by name (the first 255 characters of the SQL query). This ability to locate by name is cool because it lets you toss away lots of heritage code! I can locate an existing result set by name and let the RDO find it for me instead of my code tracking the result set.

RDO.rdoResultset and DAO.Recordset

These objects not only have names that sound alike; they have almost the same set of arguments and very similar behaviors. To understand the difference, think of a DAO.Recordset as a set of records: Each member of the set is a direct pointer to physical records in the database (not always true, but close). An RDO.rdoResultset object is the set of values that these records' fields returned. If you are reading data, there is little difference. If you are updating data, things are different. The DAO places a lock on the record at the moment you execute the DAO.Edit method. This lock remains on the record until the application executes DAO.Update. The RDO does not place a lock on the record. The RDO places a copy in a buffer when you execute RDO.Edit. When the RDO updates the record, the RDO compares the copy in the buffer to the copy existing on the database server. If the copy is unchanged, the RDO

locks the record and updates the record on the server. This difference is why the RDO has the LockType property on a result set. This is a very rough description of the general difference. For a technically correct description, you need to read the documentation for each of the many options very carefully.

The bottom line is that the RDO can often allow heavier loads on a server than the DAO because of the differences in locking mechanisms.

RDO.rdoPreparedStatement and DAO.QueryDef

These objects provide similar base functionality, but the RDO adds the extra functionality that was available only from the ODBC API or the VBSQL API. Both support the Execute method, but only RDO.Execute allows asynchronous execution of the query. (Similarly, RDO.OpenResultset can execute asynchronously, while DAO.OpenRecordset executes synchronously.) Asynchronous execution is more challenging because it means that the developer must code for reentrancy. The RDO.rdoPreparedStatement shines with stored procedures. Unlike the DAO, the RDO allows you to obtain output parameters and result codes from stored procedures. Similarly, you can set the severity level for fatal errors from the database. Another feature of RDO.rdoPreparedStatement is that the rdoParameter collection allows execution of stored procedures with different parameters without rebuilding the string by using concatenation.

In short, all of the ODBC and VBSQL features and performance that required you to code direct API calls is now available in the RDO. You do not have to create and debug a 32-bit version of the API calls when you move to Visual Basic 4.0, nor maintain both 16-bit and 32-bit versions of API code.

For the API Addict . . .

The RDO exposes three important handles to the ODBC driver for those programmers that need to make a few API calls in an application to feel fulfilled. The RDO.rdoEnvironment object exposes the ODBC environment handle, RDO.hEnv. The RDO.rdoConnection object exposes the ODBC connection handle, RDO.hDbc. Both RDO.rdoPreparedStatement and RDO.rdoResultset objects expose the ODBC statement handle, RDO.hStmt. This is one of the strengths of the RDO—the ability to code with the simplicity of the DAO, while providing easy access to the ODBC layer when needed.

It's Time to Explore!

At the beginning of this article, I gave preference to the RDO over the DAO, but I qualified it by stating that the environment can affect performance. The IntroRDO sample application allows you to time various configurations using the DAO and the RDO to determine which configuration is best for you. The DAO used in Visual Basic 4.0 is significantly faster than in Visual Basic 3.0, and the RDO is faster than either of them.

IntroRDO may teach you a lot about tuning your application in your environment because you can take a database and a query and play with all the many ways of getting your data. You can record the results to a log file for later analysis. If you are in a project management position, you may wish to make these logs into part of the project review process.

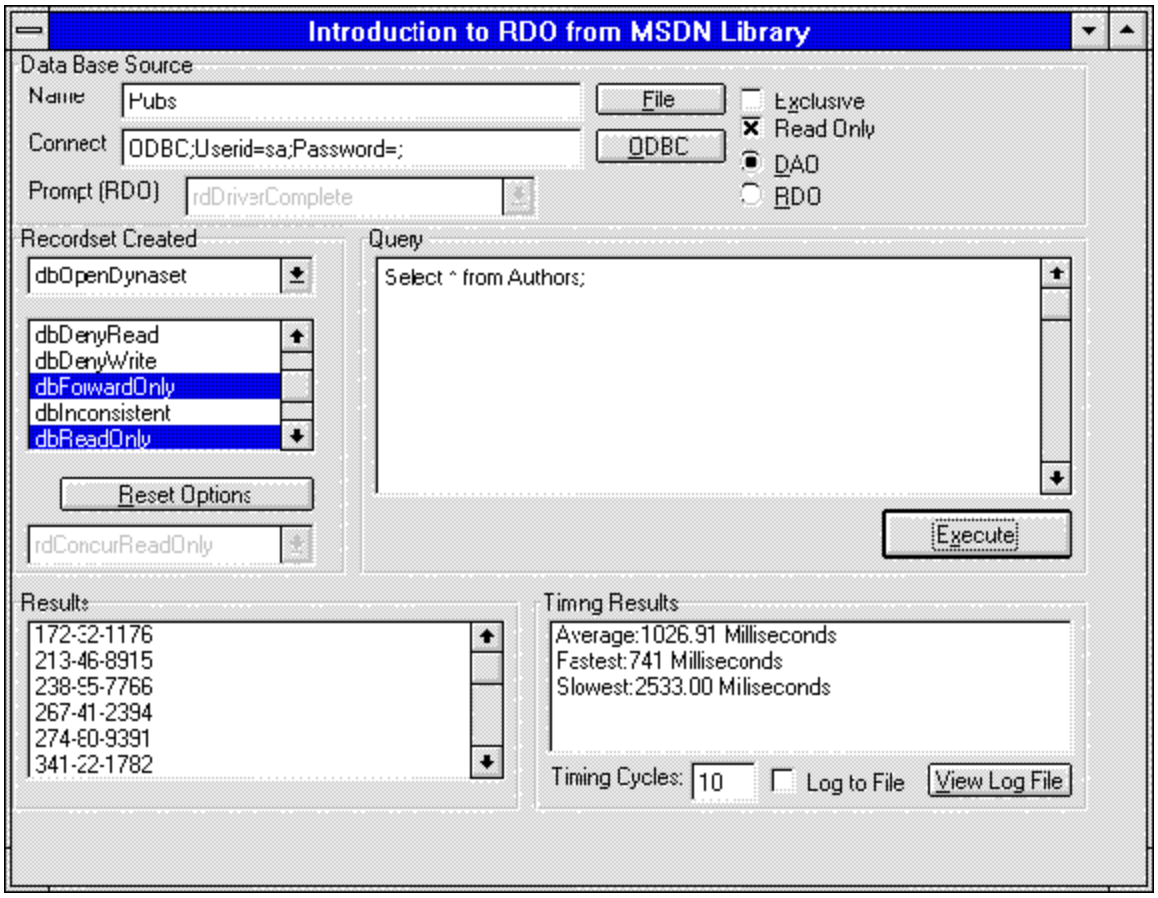

Figure 3. The various options on the IntroRDO form

The log file (Table 4) records the data in a comma-delimited format that you can easily import into Microsoft Excel or Microsoft Access for analysis. IntroRDO records times in milliseconds.

Table 4. An Example of the Generated Log File (Some Columns Deleted)

Object	Average Time	Min Time	Max Time	RSet Type	Options	Type of Access
DAO	204	180	270	dbOpenTable		Read/Write
DAO	207	180	270	dbOpenSnapShot		Read/Write
DAO	208	180	260	dbOpenTable	dbForwardOnly + dbReadOnly	Read Only
DAO	209	180	260	dbOpenSnapShot		Read/Write
DAO	215	180	271	dbOpenTable		Read/Write
DAO	890	761	1021	dbOpenDynaset		Read/Write
DAO	1039	791	1532	dbOpenDynaset		Read/Write
RDO	86	70	170	rdOpenForwardOnly		Read/Write
RDO	86	70	151	rdOpenKeyset		Read/Write
RDO	86	70	171	rdOpenStatic		Read/Write
RDO	97	80	181	rdOpenDynamic		Read/Write

If you do not find a significant performance difference between the RDO and the DAO, stick to the DAO and save conversion time. If the RDO performs significantly better, you must balance the performance improvement benefits against the conversion expense.

Where Do You Go from Here?

This article introduced you to the RDO. I believe that the RDO is the way to go for many corporate developers

because of the simplicity of a DAO-type object, the performance of direct API calls, and a very small resource footprint. Corporations can save money by cutting development hours and getting better performance from older workstations without the hardware upgrade expense. The RDO is a complex data access method with many possible configurations, so take some time to try different configurations to get the best performance before you start coding.

If you have comments you wish to share with me, drop me an e-mail message at KenL@Microsoft.Com. If you have suggestions for improving the RDO, the DAO, or Visual Basic, e-mail VBWish@Microsoft.Com.

Postscript: Licensing

Microsoft licenses the Remote Data Object for use only with the Enterprise Edition of Visual Basic 4.0. You can use it from Microsoft Excel, Microsoft Access, and Microsoft Project if you have the Visual Basic Enterprise Edition installed on the same PC. You cannot redistribute it with solution code (Microsoft Excel, Microsoft Access, and Microsoft Project). It is legal to create an OLE Automation server in Visual Basic 4.0 using RDO and to redistribute this Visual Basic server. You can use this OLE Automation server from your solution code. This introduces you to the concept of three-tier client/server solutions, the direction in which client/server solutions is moving.

Bibliography

Lassesen, Ken. "Building Add-Ins for Visual Basic 4.0."msdn_addinvb4 July 1995. (MSDN Library, Technical Articles)

Lassesen, Ken. "Mapping the Data Access Object: DAO 3.0."msdn_mapdao30 July 1995. (MSDN Library, Technical Articles)

Lassesen, Ken. "Mapping the Remote Data Object: RDO 1.0."msdn_maprdo10 August 1995. (MSDN Library, Technical Articles)

Microsoft SQL Server version 6.0 for Microsoft Windows NT, Programming SQL Distributed Management Objects. 1995. (MSDN Library, Platform, SDK, and DDK Documentation)

Microsoft Visual Basic Programming System for Windows version 4.0, Building Client/Server Applications with Visual Basic. 1995. (Ed. note: You can find an updated version of “Building Client/Server Applications with Visual Basic” in the MSDN Library, Developer Products, Visual Basic 5.0 Documentation.)

Vaughn, Bill. A Hitchhiker's Guide to Visual Basic 4 and SQL Server 6. Redmond, WA: Microsoft Press, February 1996. [This is the planned publication date.]

Vaughn, Bill. A Hitchhiker's Guide to VBSQL: The Developer's Roadmap to the Visual Basic Library for SQL Server. 3d ed. Redmond, WA: Beta V, 1994. Available via Fawcette Technical Publications at (800) 848-5523.